third clock pulse in *figure 6a*. Similarly, if the D input is 0 when the clock pulse's negative-edge occurs (the second and fourth clock pulse) output Q goes from 1 to 0.

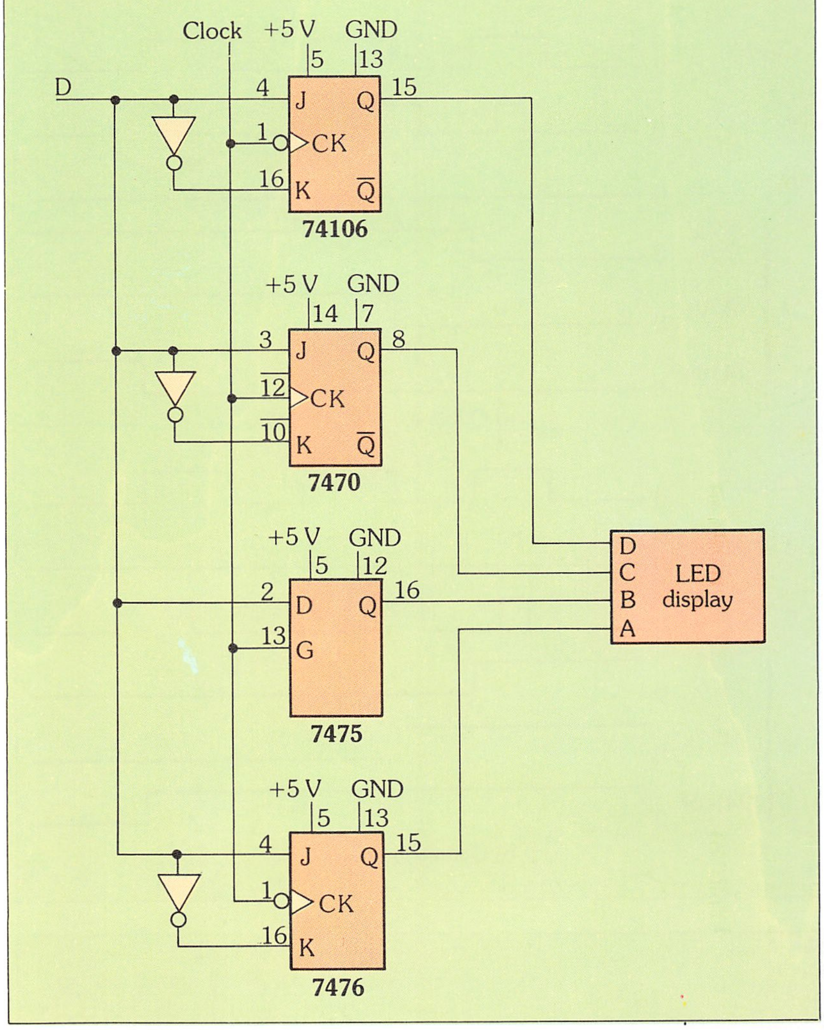
The timing diagram for *figure 6b* shows how data change must be present at the 74106-type inputs when the negative-edge of the clock pulse occurs – if no data change is present (i.e. if D is 0) then no input change occurs.

3) 7470-type latch. The 7470-type flip-flop is positive-edge triggered and so responds to input data change at the positive-edges of the clock pulses of *figure 6a*. This is identical to the 7475 latch operation in *figure 6a*. However, only one clock pulse occurs in *figure 6b* and so the 7470-type latch responds only once to the input data change on the clock pulse's positive-edge, and output Q goes to logic 1.

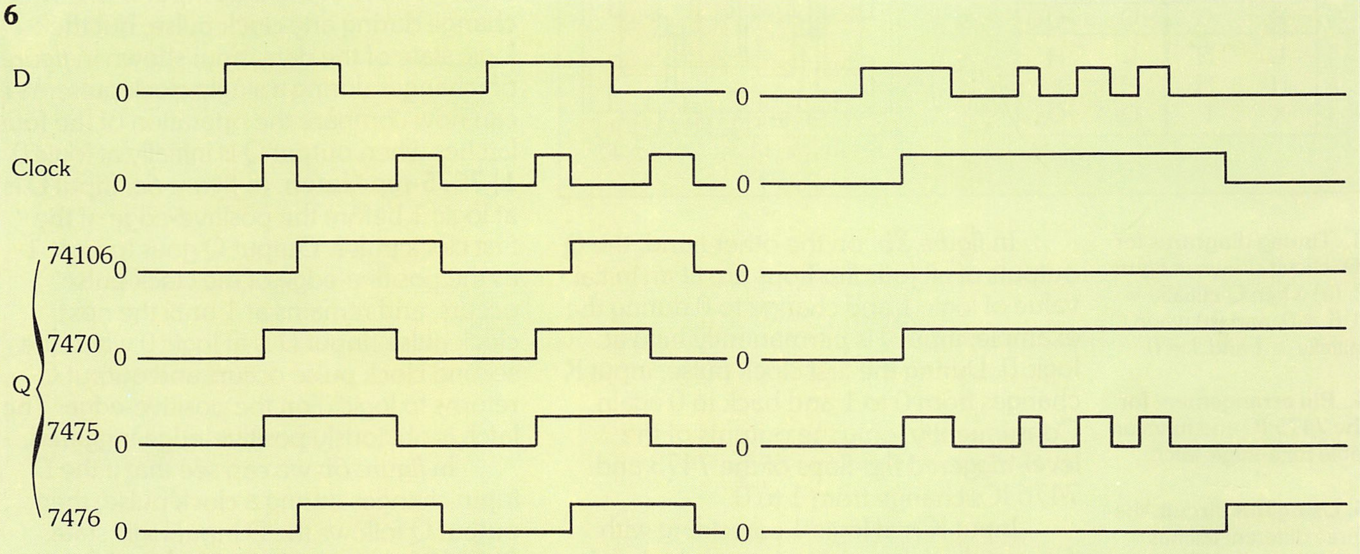
4) 7476-type latch. Being level-triggered, a latch based on the 7476-type flip-flop changes output state on the negative-edge of the clock pulse if an input data change has occurred during the clock pulse. So, in *figure 6a* the output of the 7476-type latch changes state on every clock pulse negative-edge. In *figure 6b*, although an input data change is not present at the negative-edge of the clock pulse, they have occurred *during* the clock pulse, and so the output state of the latch still changes from 0 to 1.

(continued in part 17)

5



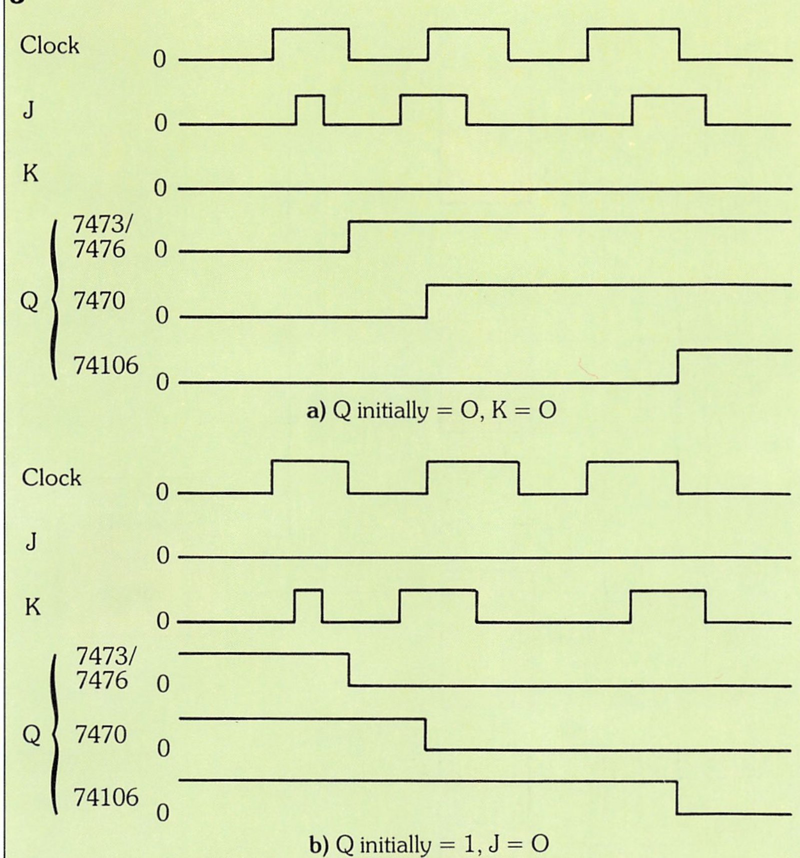
6



a) No change in data input D, during a clock pulse

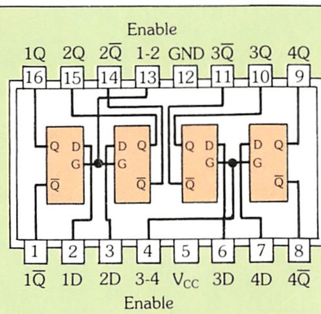
b) Change in data input D, during a clock pulse

3



4

Inputs		Outputs	
D	G	Q	$\overline{Q}$
L	H	L	H
H	H	H	L
X	L	Q_O	$\overline{Q}_O$



3. Timing diagrams for the circuit shown in figure 2: **(a)** when Q initially = 0, K = 0; and **(b)** when Q initially = 1 and J = 0.

4. Pin arrangement for the 7475 IC and function table for a single latch.

5. Using this circuit, the three different latches can be compared with the 7475-type latch

6. Timing diagram for the circuits of *figure 5*.

In *figure 3b*, on the other hand, the Q outputs of all four flip-flops are at an initial value of logic 1 and change to 0 during the example; input J is permanently held at logic 0. During the first clock pulse, input K changes from 0 to 1 and back to 0 again. Consequently, only the outputs of the level-triggered flip-flops of the 7473 and 7476 ICs change from 1 to 0.

Input K is at logic 1 coinciding with the positive-edge of the second clock pulse and so the output of the 7470-type flip-flop changes from 1 to 0 at this positive-edge.

Finally, the output of the 74106-type flip-flop changes from 1 to 0 on the negative-edge of the third clock pulse, when K is at logic 1.

Latches

The pin arrangement of the 7475 IC – known as a 4-bit binary latch – and the function table for a single latch of this IC are shown in *figure 4*. These latches are basic devices with a single data input D, and a clock input, G. Data present at input D is stored by a latch when a clock pulse is received. All flip-flops are, in fact, types of latches, but with additional facilities increasing their versatility. Generally, any type of flip-flop can be used at a latch. J-K flip-flops for example, can be used as basic latches by connecting an inverter, say, one of the six NOT gates from a 7404 hex inverter IC, between the J and K inputs. This ensures that input K of the flip-flop is always the complement of input J.

We can compare positive-edge triggered, negative-edge triggered and level-triggered latches (formed from J-K flip-flops) with the 7475-type latch, using the circuit shown in *figure 5*, where a common clock signal and a common data signal are applied to all four latches. The corresponding timing diagrams for the circuits in *figure 5* are shown in *figure 6*. These highlight the differences between the latch operations for two different conditions. In *figure 6a*, the logic state of the data input D does not change during any clock pulse, but the logic state of the data input shown in *figure 6b* changes during a single clock pulse. We can now compare the operation of the four latches when output Q is initially at logic 0.

1) 7575-type latch. In *figure 6a*, input D is at logic 1 before the positive-edge of the first clock pulse. Output Q goes to logic 1 as the positive-edge of the clock pulse occurs, and remains at 1 until the next clock pulse. Input D is at logic 0 when the second clock pulse occurs and output Q returns to logic 0 on the positive-edge. The latch is obviously positive-edge triggered.

In *figure 6b* we can see that if the D input changes during a clock pulse, then output Q follows the D input logic state.

2) 74106-type latch. Output Q of this latch goes to logic 1 if the D input is 1 when the clock pulse's negative-edge occurs. This is seen quite clearly at the first and

J-K flip-flop with clear (*figure 1b*); the 7476 dual J-K flip-flop with preset and clear (*figure 1c*); and the 74106 dual J-K negative-edge triggered flip-flop with preset and clear (*figure 1d*). The symbols ↑ and ↓ in the function tables refer to clock triggering: ↑ indicates that the flip-flop is triggered on the rising-edge of the clock pulse, i.e. it is said to be **positive-edge triggered**; ↓ indicates that the flip-flop is triggered on the falling-edge of the clock pulse i.e. it is **negative-edge triggered**; and⎍ indicates that the flip-flop is **level-triggered**.

Comparing these three triggering types of flip-flops (i.e. positive-edge triggered, negative-edge triggered, and level-triggered) as summarised in *table 1*, we see that the difference between them is evident in the instants of time when data present at the J and K inputs causes an output to occur. With the 7470-type flip-flop, data present at the J and K inputs during the positive-edge of the clock signal determines the final state of output Q. It is the input data present during the clock's negative edge which determines the final state of output Q with flip-flops like the 74106. However, changes in the input data of the 7473 or 7476-type flip-flops *while the clock signal is at logic 1* are sufficient to cause a change in the output state. In any logic block where triggering occurs when the clock is on its level state, input data change is generally restricted to immediately after reception of the clock pulse, and then only once! The circuit shown in *figure 2* is used as the basis of a comparative test between the flip-flops of the four ICs. A common clock signal is applied to all four flip-flops and the J and K inputs are common. The Q outputs of each flip-flop are then compared on a display.

The two timing diagrams which are obtained from the circuit in *figure 2* are shown in *figure 3*: these highlight the differences between the flip-flops for the two conditions shown. In *figure 3a*, the Q outputs of all four flip-flops are at an initial value of logic 0, and input K is permanently held at logic 0. During the first clock pulse – but not coinciding with either edge of the clock pulse – input J changes state from 0 to 1 and back to 0. Only the outputs of the level-triggered flip-flops change state

2

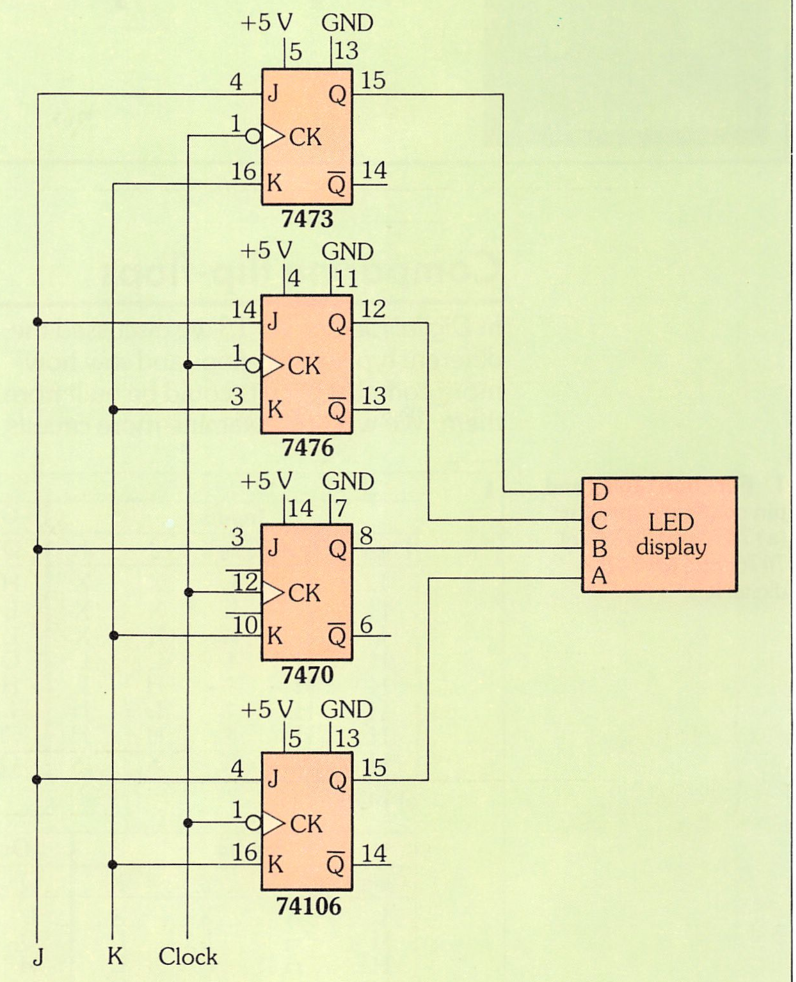


Table 1
Summary of flip-flop function tables

J-K inputs		Clock input			Output
J	K	7470	7473/7476	74106	Q
L	L	↑	⎍	↓	No change
H	L	↑	⎍	↓	H
L	H	↑	⎍	↓	L
H	H	↑	⎍	↓	Toggle

and they do so on the negative-edge of that first clock pulse.

Input J is already logic 1 at the beginning of the second clock pulse. The output state of the 7470 IC flip-flop therefore changes on this positive-edge of the clock pulse.

Output of the flip-flop from the 74106 IC changes with the third clock pulse's negative-edge, coinciding with a logic 1 at input J.

2. Circuit used as the basis of comparative tests between the flip-flops of the four ICs.

DIGITAL ELECTRONICS

14 Flip-flops and counters

Comparing flip-flops

In *Digital Electronics 13* we discussed the different types of flip-flops and saw how more complex circuits could be built from them. We will now examine more circuits.

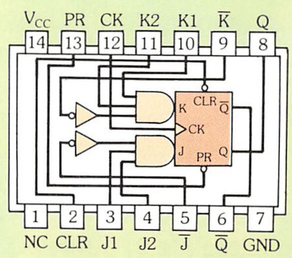
The function tables and pin configurations of the J-K flip-flops shown in *figure 1* are all for members of the 7400 series of TTL digital ICs: the 7470 AND-gated, positive-edge triggered flip-flop with preset and clear (*figure 1a*); the 7473 dual

1. Function tables and pin configurations for: (a) 7470; **(b)** 7473; **(c)** 7476; and **(d)** 74106 digital ICs.

1

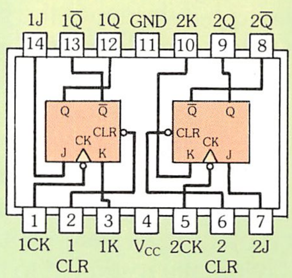
Inputs					Outputs	
Preset	Clear	Clock	J	K	Q	$\overline{Q}$
L	H	L	X	X	H	L
H	L	L	X	X	L	H
L	L	X	X	X	L	L
H	H	↑	L	L	Q_0	$\overline{Q}_0$
H	H	↑	H	L	H	L
H	H	↑	L	H	L	H
H	H	↑	H	H	Toggle	
H	H	L	X	X	Q_0	$\overline{Q}_0$

a) 7470



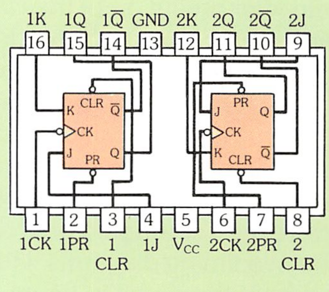
Inputs				Outputs	
Clear	Clock	J	K	Q	$\overline{Q}$
L	X	X	X	L	H
H	⎍	L	L	Q_0	$\overline{Q}_0$
H	⎍	H	L	H	L
H	⎍	L	H	L	H
H	⎍	H	H	Toggle	

b) 7473



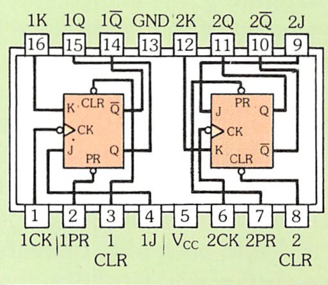
Inputs					Outputs	
Preset	Clear	Clock	J	K	Q	$\overline{Q}$
L	H	X	X	X	H	L
H	L	X	X	X	L	H
L	L	X	X	X	H	H
H	H	⎍	L	L	Q_0	$\overline{Q}_0$
H	H	⎍	H	L	H	L
H	H	⎍	L	H	L	H
H	H	⎍	H	H	Toggle	

c) 7476



Inputs					Outputs	
Preset	Clear	Clock	J	K	Q	$\overline{Q}$
L	H	X	X	X	H	L
H	L	X	X	X	L	H
L	L	X	X	X	H	H
H	H	↓	L	L	Q_0	$\overline{Q}_0$
H	H	↓	H	L	H	L
H	H	↓	L	H	L	H
H	H	↓	H	H	Toggle	
H	H	H	X	X	Q_0	$\overline{Q}_0$

d) 74106



late the average of N grades. The identification section is used to comment on and describe the program.

The environment section specifies the particular I/O requirements. This is done as a separate section because so many business problems involve the input and output of large volumes of data.

The data and procedure sections are self-explanatory and along with the identification division can be used on any COBOL operating computer. The environment section, however, has to be rewritten for different computer systems. Compare the COBOL program shown to the others that we have looked at.

Glossary

BASIC	an acronym for Beginners All-purpose Symbolic Instruction Code. A high-level programming language developed to provide an easy-to-use interactive language for computer systems
COBOL	a high-level programming language used primarily for business applications
FORTRAN	a high-level programming language used primarily for scientific applications
global variable	variables, which have been defined before procedures are called, when programming with Pascal
local variable	variables within a procedure which are completely independent of those outside the procedure. Used when programming in Pascal
loop	a sequence of programming instructions which is repeatedly executed within a programme
nesting	the process of including complete sequences of programming instructions within others
Pascal	a high-level language developed to teach good programming methods. It is a multipurpose language and enables structured programming
relational expression	a program expression relating, or comparing, variables within the program, e.g. $A = B$, $C > B$
remark, comment	a statement which enables the programmer to make comments about the program, which does not interfere with the program execution, but allows the program to be labelled and documented

.EQ. for equal to
.NE. for not equal to
.GT. for greater than
.GE. for greater than or equal to
.LT. for less than
.LE. for less than or equal to

FORTRAN uses the following control statements to direct program execution:

GO TO S_1
DO nnnnn I = integer 1, integer 2
IF (arithmetic expression) S_1, S_2, S_3
IF (logical expression) statement

The first is an unconditional transfer of control statement; the second is used to make a loop. The last two are used to make conditional control transfers.

COBOL

COBOL (**c**ommon **b**usiness **o**riented **l**anguage) was first developed in 1960 as COBOL-60 for business applications which involve processing large volumes of data, and producing reports on this data.

Each COBOL program is divided into four specific divisions or sections:

IDENTIFICATION – program identification
ENVIRONMENT – equipment description
DATA – format or form of the data
PROCEDURE – processing steps

These can be seen in *figure 4*, which is the COBOL version of our program to calcu-

4. Each COBOL program is divided into four divisions, as shown here.

4

```
     IDENTIFICATION DIVISION.
     PROGRAM – ID. AVERAGE.
     * THIS PROGRAM COMPUTES THE AVERAGE OF N
     INTEGER VALUES.
     ENVIRONMENT DIVISION.
     INPUT – OUTPUT SECTION.
     FILE – CONTROL
           SELECT INPUT– FILE, ASSIGN TO UT-S-SYSIN.
           SELECT PRINT – FILE, ASSIGN TO UT-S-
           SYSPRINT
     DATA DIVISION.
     FILE SECTION.
     FD   INPUT-FILE
          LABEL RECORDS ARE OMITTED
          DATA RECORD IS RECORD-IN, RECORD-IN2.
     01   RECORD – IN.
          02 ELEMENT           PIC     999.
     FD   PRINT – FILE
          LABEL RECORDS ARE OMITTED
          DATA RECORD IS RECORD-OUT, RECORD-OUT2.
     01   RECORD-OUT.
          02 NUM-OUT           PIC     999.
     01   RECORD-OUT 2.
          02 AVERAGE-OUT       PIC     9999V999.
     WORKING – STORAGE SECTION.
1    77      TOT-SUM           PIC     999 VALUE 0.
2    77      FLAG              PIC     9   VALUE 0.
     77      TOT-VALUES        PIC     999.
3    PROCEDURE DIVISION.
4          OPEN INPUT INPUT-FILE, OUTPUT PRINT-FILE.
5          READ INPUT-FILE AT END MOVE 9 TO FLAG.
7          MOVE ELEMENT TO TOT-VALUES.
8          READ INPUT-FILE AT END MOVE 9 TO FLAG.
10         PERFORM B100 UNTIL FLAG = 9.
11         DIVIDE TOT-VALUES INTO TOT-SUM GIVING
           AVERAGE-OUT.
12         WRITE RECORD-OUT2.
13         CLOSE INPUT-FILE, PRINT FILE.
14         STOP RUN.
     B100.
15         MOVE ELEMENT TO NUM-OUT.
16         WRITE RECORD-OUT.
17         ADD ELEMENT TO TOT-SUM.
18         READ INPUT-FILE AT END MOVE 9 TO FLAG.
     B900-RETURN.
```

FORTRAN

FORTRAN (**for**mula **tran**slation) was one of the first high-level languages and was developed primarily for scientific applications. FORTRAN and COBOL between them accounted for most of the programs written in the 50s and 60s.

To introduce FORTRAN, let's look at a FORTRAN version of our program to calculate the average of N grades:

```
1  C     FORTRAN PROGRAM TO COMPUTE
2  C     AVERAGE OF N GRADES
3        READ (5, 10) N
4  10    FORMAT (I2)
5        SUM = 0
6        DO 20 I = 1, N
7        READ (5, 15) V
8  15    FORMAT (F5.0)
9  20    SUM = SUM + V
10       AVERAG = SUM/N
11       WRITE (6, 30) AVERAG
12 30    FORMAT (16H THE AVERAGE
         IS, F6.2)
13       END
```

The line numbers on the left hand side are for reference only, but the statement numbers that occur to the right of these are needed by the program.

You may have gathered that lines 1 and 2 are comments, and this is indicated by a C in the second column. FORTRAN statements have to obey very strict rules over where they can be placed on the coding line. The statement numbers, by the way, must be located in columns 1 to 5; the statement itself must not begin until column seven. In BASIC and Pascal it doesn't matter where statements begin, however in FORTRAN this is important.

Line 3 is the first executable statement. It is a READ statement, much like those in other languages, except for the bracketed numbers. The bracketed 10, references statement number 10 (line 4) while the 5 tells the computer that the input data will come from logical unit 5 – a terminal – which in this case is the same device the program was read from (see *figure 3*). Statement 10 tells the computer the form or FORMAT of the data value to be read in. In this case, the number will be an integer – indicated by the I – and it will be found in the first two columns of data. This allows us to use any integer number from 1 to 99. If we knew that the number was to be 9 or less for example, then we would indicate this by the format description, I1.

In line 5, SUM is assigned to zero in the same way as in the other languages.

The statement at line 6:

DO 20 I = 1, N

is used for looping, and is similar in operation to Pascal and BASIC FOR statements. It directs the computer to execute all the instructions that follow (down to and including statement 20), N times. This is called a **do loop** in FORTRAN. As before, the variable I is used to count the number of times the loop has been executed. After statement 20 (line 9), I will be incremented by 1, and if greater than N, the statement at line 10 will be executed. Otherwise, control is returned to line 7.

The statement at line 7 is another READ but this time FORMAT (F5.0) is used. This tells the computer to read the grade from the first five columns of data, and that it is to have a decimal point assigned. The grades are added together at line 9, while line 10 is used to compute the average. The variable name is limited to six characters, as it was in BASIC.

The answer is printed at line 11, and this time logical unit 6 is used – the printer. The format of the output is specified by the statement at line 12. Beginning at the first column of the output page, it will print:

THE AVERAGE IS XXX.XX

The answer is to be given as a decimal number with the decimal fraction to two places. The total number of digits plus the decimal point is not to be greater than 6. The END statement ends the program.

Other FORTRAN features

Variables used in FORTRAN (like those in BASIC or Pascal) may be integer, real or logical. Double precision working can be used, which makes the word size twice as big and enables the programmer to work with more complex numbers. Variables are assumed to be integer if the variable name begins with any of the letters from I to N. Otherwise they will be in floating point form. Any variation from this has to be declared with a TYPE statement.

The arithmetical operators are the same as those in BASIC or Pascal, but logical operators must be of the form:

```
    END ;
  AVERAGE = SUM/N
  END ;
PROCEDURE PRINTAVERAGE ;
  BEGIN
    WRITELN ('THE AVERAGE IS', AVERAGE);
  END ;
```

Notice how this program closely resembles the first level statements discussed earlier, i.e. of describing the problem:

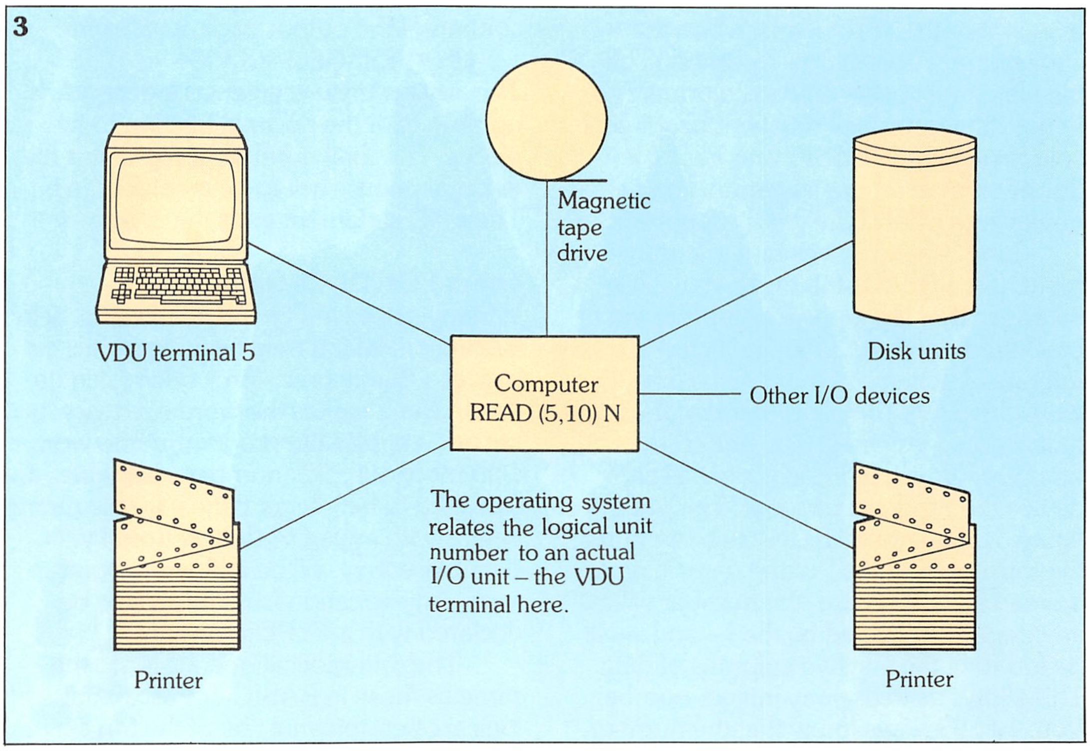
1.0 Read Numbers
2.0 Compute Average
3.0 Print Answer

Notice also that the variables N and AVERAGE are global variables – defined at the beginning and available to be used by any following procedures that reference them. Remember though, that variables declared within the procedure, such as SUM AND I, are only available within the COMPUTE AVERAGE procedure.

Pascal also allows the programmer to declare arrays. This is done at the same time as variables are declared and takes the form shown in the fifth line below. We can now make our program similar to the top down first level statements, except that the procedures must be defined before they are referenced by other statements:

```
PROGRAM AVERAGE (INPUT, OUTPUT);
  (* COMMENTS SAME AS BEFORE *)
  VAR N,I : INTEGER ;
    AVERAGE : REAL ;
    V : ARRAY (1..50) OF REAL ;
PROCEDURE READNUMBERS ;
  BEGIN
    READ (N) ;
    FOR I : = 1 TO N DO
    READ (V[I]) ;
  END ;
PROCEDURE COMPUTEAVERAGE ;
  VAR SUM : REAL ;
  BEGIN
    FOR I : = 1 TO N DO
      BEGIN
        SUM : = SUM + V(I);
      END ;
    AVERAGE : = SUM/N ;
  END ;
PROCEDURE PRINTANSWER ;
  BEGIN
    WRITELN ('THE AVERAGE IS', AVERAGE) ;
  END ;
  BEGIN
    READNUMBERS ;
    COMPUTEAVERAGE ;
    PRINTANSWER ;
  END.
```

The main difference between this and the previous version is the use of the array, V, declared for 50 values. The grades are read in the first procedure READ NUMBERS; the average computed in the second, COMPUTE AVERAGE; and the answer printed in the third, PRINT ANSWER. So, you can see how useful Pascal is, for top down structured programming.



3. The program specifies which device has the data with a logical unit number (5 in this case).

have:

```
11  for I : = 1 TO N DO
12      BEGIN
13          READ (V) ;
14          SUM : = SUM + V;
15      END ;
```

The block of statements bounded by the BEGIN and END keywords will be executed N times. The variable I is incremented each time the END keyword is reached. Once incremented, it is tested and if the result exceeds N, the next statement following this block (line 16) is executed.

Another construct which permits looping is the WHILE. The statements that read and sum the V values could have been executed in the following way:

```
WHILE I < = N DO
   BEGIN
      READ (V) ;
      SUM : = SUM + V ;
      I : = I + 1 ;
   END ;
```

The WHILE statement is recommended for structured programming. The block of statements bounded by BEGIN and END are executed while I is equal to or less than N. I is tested first, and if the condition in the expression is not satisfied, then the block of statements is not executed. Notice that the WHILE loop is different from the FOR loop, as the FOR loop is always executed at least once because testing is not done until the block has been processed.

Returning to our full program, the average is computed and printed by the statements at lines 16 and 17.

Additional Pascal statements

Although Pascal does not provide as many statement types as other languages, it can still be used to write well-structured programs. You will have noticed that the program blocks are written in a zig-zag format. This has no effect on the program's execution, but is done simply to make each program block more readable.

Blocks are one or more statements contained within a BEGIN and END keyword, and an entire block is treated as a single statement. Blocks can also have other blocks **nested** within them. A (third) control structure, useful for decision making processes is the IF THEN, ELSE structure. This takes the general form:

```
IF expression THEN
(statement or block)
ELSE
(statement or block)
```

IF the expression is true, THEN the first block is executed or, ELSE the second block is executed (i.e. if the expression isn't true). You'll remember from the discussion of structured programming that these, the constructs:

FOR,
WHILE, and
IF THEN ELSE,

and sub-routines are all that are needed.

In Pascal, sub-routines are referred to as **procedures**. These not only provide a means of reducing the total amount of coding required, but also allow independent modules to be defined and modular programming to be implemented. Pascal procedures are similar to BASIC sub-routines, with a few key differences.

The most important difference is the ability to declare **local variables** within a procedure. This simply means that new variables can be declared within a procedure, and these are completely independent of those outside it. On the other hand, **global variables** are variables that have been defined before procedures are called, and which can be referenced directly within each procedure or module. We would redefine our program as follows:

```
PROGRAM AVERAGE (INPUT, OUTPUT) ;
   (* COMMENTS SAME AS BEFORE *)
   VAR N : INTEGER ;
      AVERAGE : REAL ;
   BEGIN
      READN ;
      READANDSUM ;
      PRINTAVERAGE ;
   END ;
```

A procedure for each READ N, READ AND SUM and PRINT AVERAGE could then be written:

```
PROCEDURE READN ;
   BEGIN
      READ(N) ;
   END ;

PROCEDURE READANDSUM ;
   VAR V, SUM : REAL ;
      I : INTEGER ;
   BEGIN
      SUM : = 0 ;
      FOR I : = 1 TO N DO
         BEGIN
            READ (V) ;
            SUM : = SUM + V ;
```

Pascal

Pascal was developed by Professor Niklaus Wirth in the 1970s to teach good programming style to computer science students. The language itself was named after the French mathematician Blaise Pascal (since it is a name, not an acronym, only the first letter is capitalised). Although originally designed for teaching, Pascal has rapidly become one of the most popular languages; one of its major advantages being that it does not use many different language constructs or statement types. Although some Pascal compilers do not employ some of the more useful facilities, such as data statements, it is still an extremely practical language for scientific, personal and systems programming. Pascal has also been used on many small mini and microcomputers systems *because* of this lack of varied statement types.

The first noticeable difference between Pascal and BASIC is that in Pascal, all variables used must be defined at the beginning of the program: this is done with a VAR statement. Variables can be specified as either integer or real (floating point). To take as an example our problem of computing the average of N values, we need to define I and N, as well as the variable V. This is done with the following statements:

```
VAR I, N : INTEGER ;
SUM, AVERAGE , V : REAL ;
```

I and N are integers, while SUM, AVERAGE and V are real numbers. Notice the seven character word AVERAGE: Pascal allows variable names of up to eight characters.

Let's look at the Pascal program that calculates the average of N numbers, and compare it to our earlier BASIC program.

```
1.  PROGRAM AVERAGE (INPUT, OUTPUT);
2.    (* THIS PROGRAM COMPUTES THE
      AVERAGE OF N GRADES *)
3.    (* N = NUMBER OF GRADES *)
4.    (* V = CURRENT VARIABLE OR GRADE
      READ *)
5.    VAR I , N : INTEGER ;
6.      SUM, V, AVERAGE : REAL;
7.    BEGIN
8.    READ (N) ;
9.    I : = 1 ;
10.     SUM : = 0 ;
11.   FOR I : = 1 TO N DO
12.     BEGIN
13.       READ (V) ;
14.       SUM : = SUM + V ;
15.     END ;
16.   AVERAGE : = SUM/N ;
17.   WRITELN ('THE AVERAGE IS' , AVERAGE) ;
18.   END.
```

The line numbers in Pascal programs are only used for reference in describing the program, they are not used by the program as the statement numbers are in BASIC.

The first statement in the program:

```
PROGRAM AVERAGE (INPUT, OUTPUT);
```

specifies the name of the program, its beginning and that it will use both input and output in its execution. This statement may vary in style. Pascal statements end with a semicolon.

Lines 2, 3 and 4 are comments, denoted by the opening and closing parenthesis with asterisks. Comments can be placed anywhere on any line, and do not require a semicolon as they are not statements.

Lines 5 and 6 contain the statements that declare the variables, and these are followed by the main body of the program. This set of statements is initiated by the BEGIN statement and terminated by the END statement. This final END statement differs from the others (like the one that terminates the loop in our program) as it is followed by a full stop.

Line 8 is used to read in the number of grades to be averaged, N. Statements at lines 9 and 10 are used to assign values to the variables I and SUM. Notice that these assignments are made with the colon and equals sign:

```
9  I : = 1 ;
10  SUM : = 0 ;
```

The general form of the assignment expression is:

```
identifier : = expression ;
```

and of course the identifier is a variable that has to have been previously declared.

Pascal provides a number of statements that control program flow (constructs). Remember in BASIC there are three control statements:

```
IF expression THEN nnnnn
GO TO nnnnn
FOR variable = expression TO expression
NEXT variable
```

Pascal gives us the equivalent of these statements and several more. In place of BASIC's FOR – NEXT construct, we now

program. This is the reason why sub-routines must end with a RETURN statement.

Sub-routines are called for in BASIC by a statement that takes the form:

line number GO SUB nnnn

where nnnn is the first statement number in the sub-routine. *Figure 2* illustrates how sub-routines are generally used: two are shown, beginning at lines 1000 and 2000.

We could have used a sub-routine in our average grade program. Consider this:

```
 10 PRINT "ENTER NUMBER OF GRADES"
 20 INPUT N
 30 GO SUB 60
 40 REM OTHER PROGRAM STATEMENTS
 50 STOP
 60 FOR I = 1 TO N
 70 PRINT "ENTER V"
 80 INPUT V
 90 NEXT I
100 RETURN
```

Once the total number of grades has been read in statement 20, the sub-routine at statement 60 is called to read in the individual grades. It does this N times to record all the grades, returns to statement 40 (which stands for the remaining program statements that calculate the average) then goes to statement 50 and stops.

You will have noticed that statements 60 and 90 loop until I is equal to N. BASIC provides two statements, FOR and NEXT, that are used to implement loops, and they are used as follows:

```
FOR variable =
expression 1 TO expression 2
STEP expression 3
.
.
.
.
.
NEXT variable
```

So we can rewrite our earlier program as:

```
 10 DIM V(100)
 20 PRINT "ENTER NUMBER OF GRADES"
 30 INPUT N
 40 SUM = 0
 50 FOR I = 1 TO N
 60 PRINT "ENTER GRADE"
 70 INPUT V(I)
 80 SUM = SUM + V(I)
 90 NEXT I
100 PRINT "THE AVERAGE IS"; SUM/N
120 STOP
```

We have added statements 50 and 90. Statement 50 directs the computer to process all the statements down to 90, N times. Each time statement 90 is reached, I will be incremented by 1 and if the result is not greater than N, the loop will be repeated from statement 50. Once I is greater than N, statement 100 will be executed. Notice that in statement 100, the division for SUM/N is performed in the PRINT statement. This saves one line of code and the memory location previously known as AVERAG.

The increment that is added to I in statement 90 is one. However, we could change this increment or step size by specifying it in the FOR statement. A step size of two would be given by:

```
50 FOR I=1 TO N STEP 2
```

which will execute statements 50 to 90 N/2 times.

Some basic interpreters provide the "IF THEN ELSE" feature. This control structure, as discussed in the previous chapter, helps to write programs in a structured fashion. Unfortunately, however, this feature alone is not enough for good structured programs and BASIC is not one of the languages typically recommended for implementing the structured programming technique.

Below: programs can be written to be as friendly as possible to the user. Here, special screen formats have been programmed to make data input easier. (Photo: STC Business Systems Ltd).

You may be wondering why we left the 'E' off the variable name AVERAG. This is because most BASIC interpreters only permit variable names to be a maximum of 6 characters.

Returning to our program, you can see that statement 130 displays the result of our program and statement 140 directs the computer to stop.

Additional BASIC features

If we want to preset the grades for our program, rather than reading them in from the terminal one be one, we can use the DATA statement. This takes the form:

line number DATA data list

To set the values to the data list, a READ statement is used. This takes the form:

line number READ variable list

So, let's assume that the grades we want to average are 82, 47, 95, 100, 68 and 91. Our program will thus be written:

```
10  REM THIS PROGRAM COMPUTES
    AVERAGE OF N NUMBERS
30  READ V1, V2, V3, V4, V5, V6
80  SUM = V1 + V2 + V3 + V4 + V5 + V6
120 AVERAG = SUM/6
130 PRINT "THE AVERAGE IS"; AVERAG
140 STOP
150 DATA 82, 47, 95, 100, 68, 91
```

2. General usage of sub-routines.

2

BASIC	GENERAL FORM
REM BEGIN	(*BEGIN*)
.	.
100 GO SUB 1000	CALL SUB A
.	.
200 GO SUB 2000	CALL SUB B
.	.
300 GO SUB 1000	CALL SUB A
.	.
400 GO SUB 2000	CALL SUB B
.	.
500 STOP	END
1000 REM SUB-ROUTINE A	SUB-ROUTINE A
.	.
1099 RETURN	RETURN
2000 REM SUB-ROUTINE B	SUB-ROUTINE B
.	.
2099 RETURN	RETURN

Statement 30 will read the values of the data items specified in statement 150. The variable V1 is assigned the value 82, V2 is assigned 47 and so on. The DATA and READ statements are useful for presetting variables. If we had to change the six values to six other grades only line 150 would have to be altered.

We could have used direct assignments for the variables V1 to V6:

```
12 V1 = 82
14 V2 = 47
16 V3 = 95
```

and so on, but then making any change would involve changing the value in each statement. Using a DATA statement is, however, still not as flexible as the first method. If we were to change the number of grades to be averaged we would have to change the number of Vs in the READ statement.

Another method of solving this problem is to define a table or an array of variables. This is basically a collection of variables arranged in such a way to enable them to be more easily used in the program (see *Basic Computer Science 6*). It is necessary to use a DIM (dimension) statement to declare an array. For example:

```
DIM V (50)
```

instructs the computer to set aside 50 memory locations for the one dimensional array V.

To read all the values from the array V, the following statements can be used:

```
20 for T = 1 TO 50
21 READ V (T)
22 NEXT T
```

and all the values in the array can be used.

Sub-routines

Like other languages, BASIC permits the programmer to write and use sub-routines which are generally used when a series of instructions needs to be executed a number of times. Sub-routines are called by a statement in the main program and after the statements in the sub-routine are completed, control is returned to the main

interpreter not to leave any designated spaces between the last character in quotations and the most significant digit of the value of AVERAGE.

When the statement 30 PRINT A,B was typed in, a comma was used between the A and B. This comma is the instruction to display the value of A (4) in the second character position, and the value of B (9) in the sixteenth position. If the statement, 30 PRINT A; B had been written, however, the value of A would still be in the second character position, but the value of B would be printed in the fifth position; the comma and semicolon punctuation marks therefore determine the positions of characters to be displayed or printed. This example is specific to one particular version of BASIC; other print positions can be specified for other BASICs either with different punctuation marks or by different means.

A program in BASIC

```
 10 REM THIS PROGRAM COMPUTES AVER-
    AGE OF N NUMBERS
 20 PRINT "ENTER NUMBER OF ITEMS"
 30 INPUT N
 40 SUM = 0
 50 I = 1
 60 PRINT "ENTER GRADE"
 70 INPUT V
 80 SUM = SUM + V
 90 IF I = N THEN 120
100 I = I + 1
110 GO TO 60
120 AVERAG = SUM/N
130 PRINT "THE AVERAGE IS"; AVERAG
140 STOP
```

As you can see from this simple program there are some commands that we have not yet discussed, for instance line 10:

```
10 REM THIS PROGRAM COMPUTES
   AVERAGE OF N NUMBERS
```

This **remark** (REM) or **comment statement** enables comments to be made about programs. This is an extremely useful way of labelling (or documenting) programs to aid the programmer in remembering what they were for and how they work. Although REM statements are not executed they are printed out when the program is listed.

Statement 20 prompts us to enter the number of grades, and statement 30 instructs the computer to wait for this number to be entered from the keyboard and then assigns it to the variable N. Statement 40 zeros the variable SUM (as we saw in *Computer Science 8*) and statement 50 sets I to 1. Statement 60 prompts us to enter the grade; statement 70 (like statement 30) instructs the computer to wait for input and assigns the grade entered to the variable V; statement 80 accumulates the grades.

Another command that we have not yet seen is contained in statement 90. Remember in *Computer Science 8* we found that we needed a test to exit from a loop. The IF statement provides this, and takes the form: line number IF relational expression THEN statement number. The **relational expression** comprises variables and *relational* operators, rather than arithmetic operators, such as:

= Equal to
> Greater than
< Less than
>= Greater than or equal to
<= Less than or equal to
>< or <> Not equal to

The IF statement transfers control to the statement number following the word THEN, if the expression is true. If it is not true, then the statement following the IF statement will be executed. In our example program, IF the value assigned to I is equal to the value assigned to N, THEN statement number 120 will be executed. If not, statement 100 (which follows 90) will be executed.

Statement 110, on the other hand, is a direct or unconditional transfer of control instruction. When it is executed, control is passed directly to the specified statement number. Here, control returns to line 60, thus making the **program loop**. When I is less than 6, one will be added to I at statement 100 and statement 110 will direct the computer to loop back and execute that part of the program again from line 60. This enables us to enter the six grades, which is why we set N to 6.

When I = 6, all the grades have been added together and we are now ready to compute the average. So when I = N, statement 90 directs the computer to execute statement 120. Here the variable AVERAG is assigned the value of the accumulated grades (in SUM) divided by the number of grades N.

```
A = 16/9
```

divides 16 by 9 and stores the result in A. In order to see the value of A, simply type PRINT A – this displays the value stored on the screen (see *figure 1*).

Each statement is given a line number (any value from 1 to n, where n is the limit of the memory or digit capacity) to ensure that the program statements are executed in order. These are often incremented in tens, i.e. 10, 20, 30 and so on, so that additional statements can be inserted between the existing statements if necessary.

Now suppose that we want to write a program that sets A to a certain value, and then later sets the variable B to equal A + 5. We type in:

```
10 A = 16/4
20 B = A + 5
30 PRINT A, B
RUN
```

The computer divides 16 by 4 and stores the result in A; 5 is then added to the contents of A and the result is stored in B; the values of A and B are then displayed. This is shown in *figure 1*. Statements with no line numbers are usually executed immediately; those with line numbers are held back for later execution, which is controlled by the RUN command.

You will have gathered by now that the symbol = is used to assign a variable to an expression. An expression is defined as some combination of operators, variables and constants. The following operators are used in BASIC:

+	addition
−	subtraction
*	multiplication
/	division
^ ↑ **	exponentiation

Variables are used for assigning specific memory locations for values that change. The value of a variable is maintained in memory: the memory contents being updated each time a new value is assigned to the variable. Variable names comprise one or more alphanumeric characters, the first character, though, must *always* be alphabetic.

If variables are used to store alphabetic characters, they must be specially identified by attaching a symbol to the end of the variable name. For example:

```
10 A$ = "THIS IS A GOOD BOOK"
```

instructs the computer to assign the necessary memory locations to contain the digital codes (such as ASCII) for the symbols or characters that follow, beginning at a memory location specified by A. The quotation marks around the character string instruct the computer to assign codes for this alphanumerical data. Blank spaces are also treated as characters, and there is a code for space. If we were to type in the statement above, followed by:

```
60 PRINT A$
RUN
```

then THIS IS A GOOD BOOK will be displayed.

Thinking back to the example in *Computer Science 8* of the students' average grade, the different program levels were:

1.0 Read numbers
1.1 Read the number of values to be averaged, N
1.1.1 Read N values of numbers, V
2.0 Compute average
2.1 SUM = SUM + V
2.1.1 Average = SUM/N
3.0 Print answer

We have seen that it is easy to carry out operations like SUM = 0 or SUM = SUM + V in BASIC, but how do we read N or V and how do we print Average?

Input to and output from the computer terminal is carried out by the INPUT and PRINT statements. To input the number N, we simply type:

```
10 INPUT N
```

When this statement is executed the computer will prompt us by displaying a question mark on the screen – it then waits until a number for N, followed by a carriage return (press the RETURN or ENTER key), is typed in from the keyboard.

To output an item we type, for example:

```
60 PRINT AVERAGE
```

Or, if we want the variable AVERAGE to be identified, we can display THE AVERAGE IS, followed by its value. In this case we would change line 60 to:

```
60 PRINT "THE AVERAGE IS"; AVERAGE
```

Remember, the quotation marks specify alphanumeric data; the semicolon tells the

BASIC

BASIC (**b**eginners **a**ll-purpose **s**ymbolic **i**nstruction **c**ode) was developed during the mid-sixties as a 'friendly' (i.e. easy to understand, learn and use) effective language for beginners and non-computer specialists. It became popular during the late 1970s and early 1980s with the spread of microcomputers; now, most **interactive systems** (i.e. systems that allow direct interaction between the user and the program) and almost all personal computers support (i.e. will run) BASIC.

There are a number of different versions of BASIC, each with slightly different features adapted to run on different machines.

BASIC statements

BASIC statements instruct the computer as to what operations are to be performed. For example, to multiply two numbers, say 3 and 4, and assign the result to the variable A (memory storage location assigned to A), you type into the keyboard:

```
LET A = 3 * 4
```

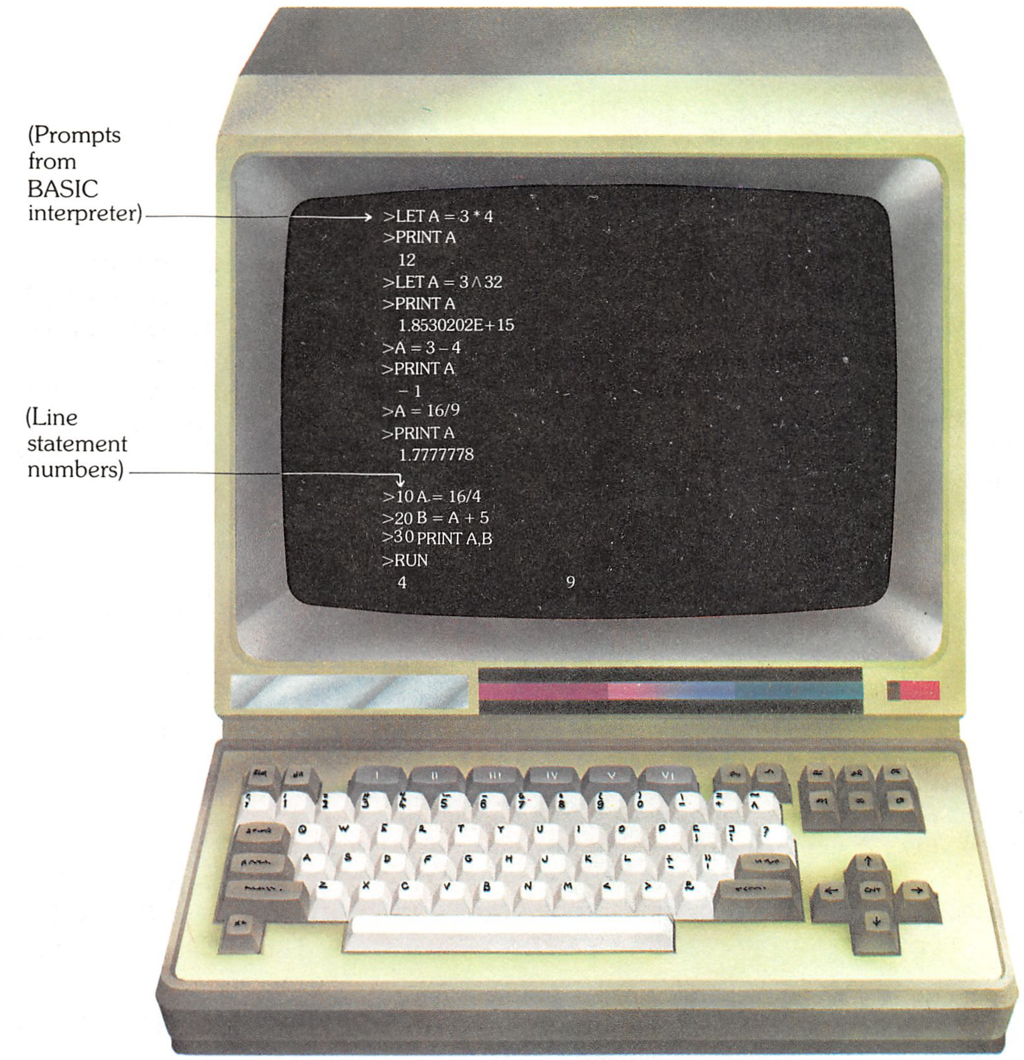
The computer then multiplies 3 and 4 and sets the variable A to 12 (the result). Similar BASIC statements permit other arithmetical functions to be carried out.

Most BASIC interpreters permit the programmer to leave out the 'LET'. For example:

```
A = 3−4
```

subtracts 4 from 3 and stores the result in A;

1



1. As you type in your program it will be displayed like this on the terminal screen.

9

BASIC COMPUTER SCIENCE

Programming languages

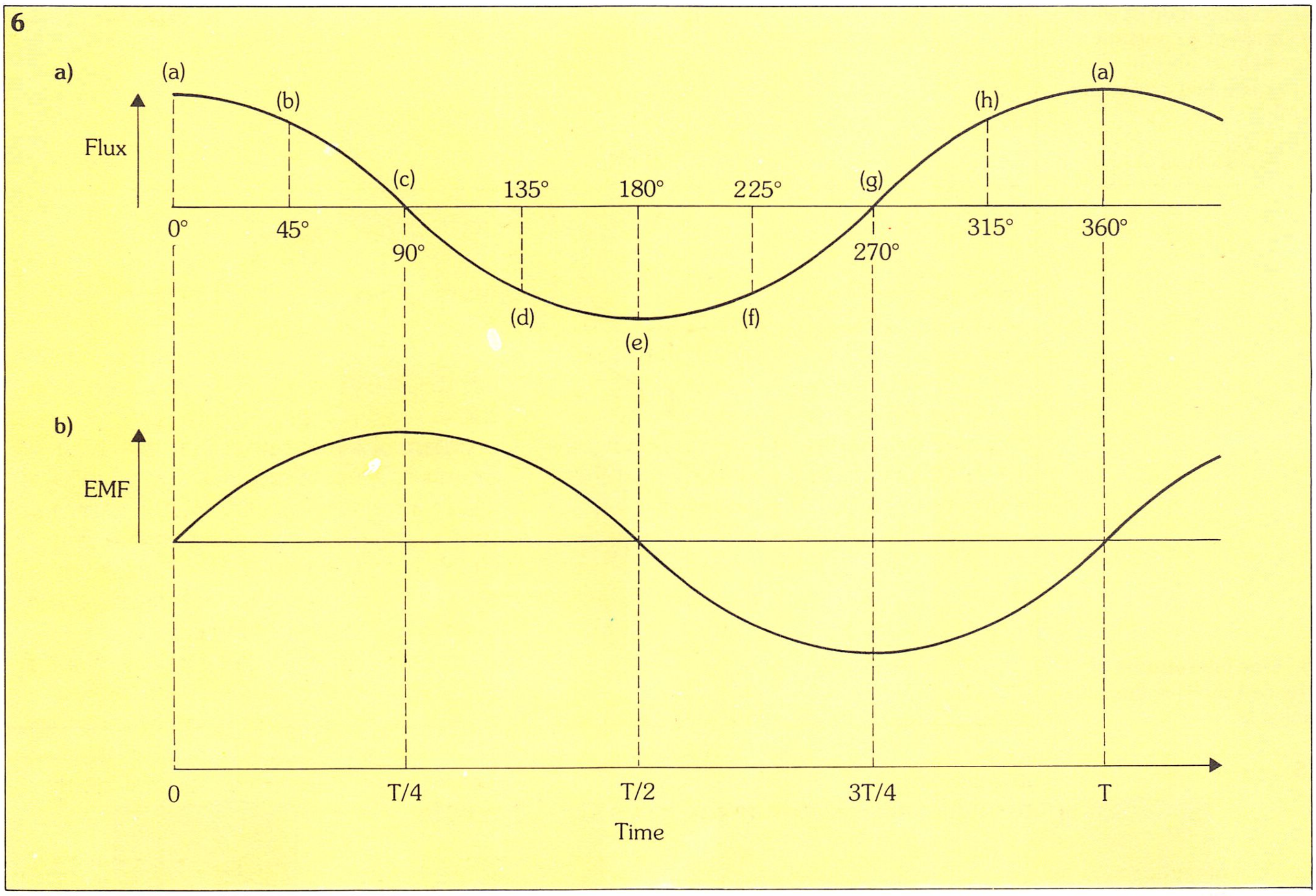
Introduction

As we have previously noted, there are three types of programming language: machine; low-level; and high-level. Machine languages, or machine code, are machine-specific and are written in binary; instructions written in this way, as you know, can be performed immediately by the computer and require no further translation. Low-level languages provide a step forward for the programmer. Although they are machine oriented languages, mnemonic codes and symbolic addresses are used in place of machine codes. These assembly languages are each closely related to a specific machine language; manufacturers provide an assembler to translate this assembly language into machine code.

High-level languages, on the other hand, were primarily developed to make the task of programming easier and are problem oriented rather than machine oriented. They fall into five broad categories: commercial (e.g. COBOL); scientific (e.g. FORTRAN); special purpose (e.g. CORAL-66); command languages for operating systems; and multipurpose languages (e.g. PL1 and Pascal).

Coding, one of the six basic steps of programming, cannot be done without a good working knowledge of the language being used and we will now go on to examine the features of four commonly used languages.

Right: computers at work in the office. Each user has their own terminal and floppy disk drive but would share resources, such as the printer, with other users. (Photo: STC Business Systems Ltd).



change of flux linked.

If this is carried out accurately, we shall obtain the EMF graph shown in *figure 6b*. This has a shape identical to the flux graph, but is displaced by 90°. When the flux linked is increasing, the generated EMF will be negative, and when the flux linked is decreasing, the EMF will be positive. When the flux is constant (at maximum or minimum) the EMF is zero.

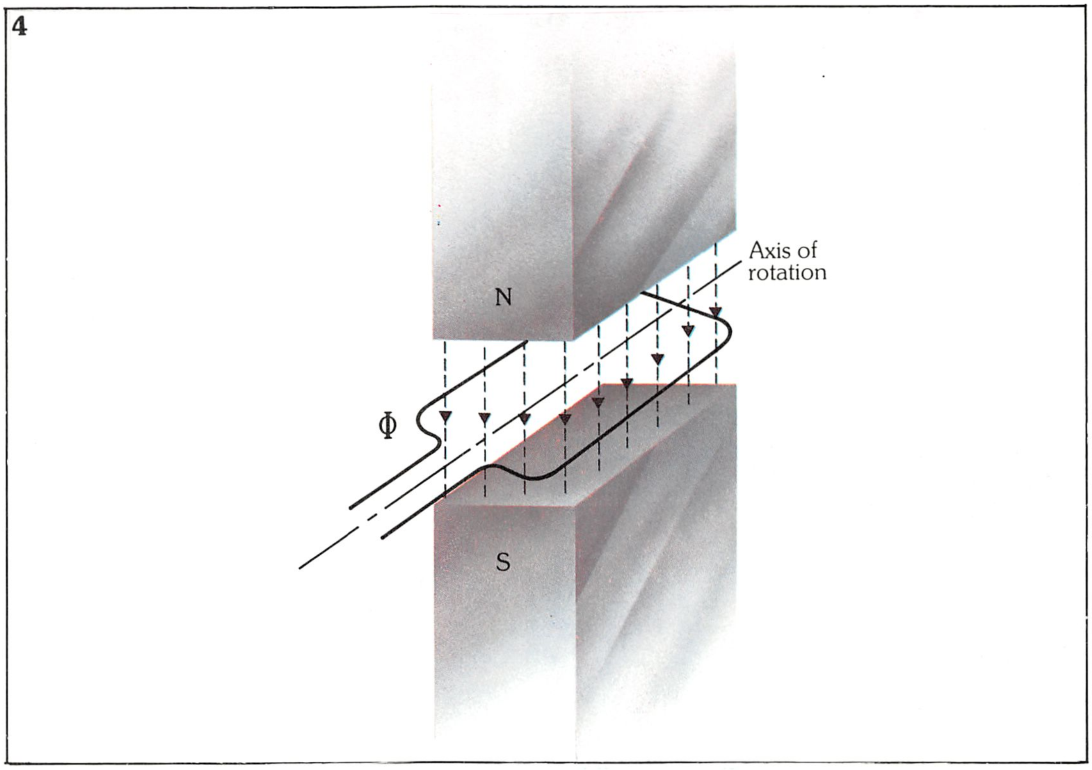
The flux and EMF curves are sinusoidal, and we shall find that voltages and currents that vary with time like this are very common. The principle behind this EMF generator forms the basis of the alternator, although, in practice, this single loop is replaced by a coil that has a great number of turns of wire. This is to generate a much larger alternating voltage than a single loop could ever practically produce. □

6. (a) Flux linkages plotted for positions *a-h* (figure 5); **(b)** EMF generated over time for the rotating coil.

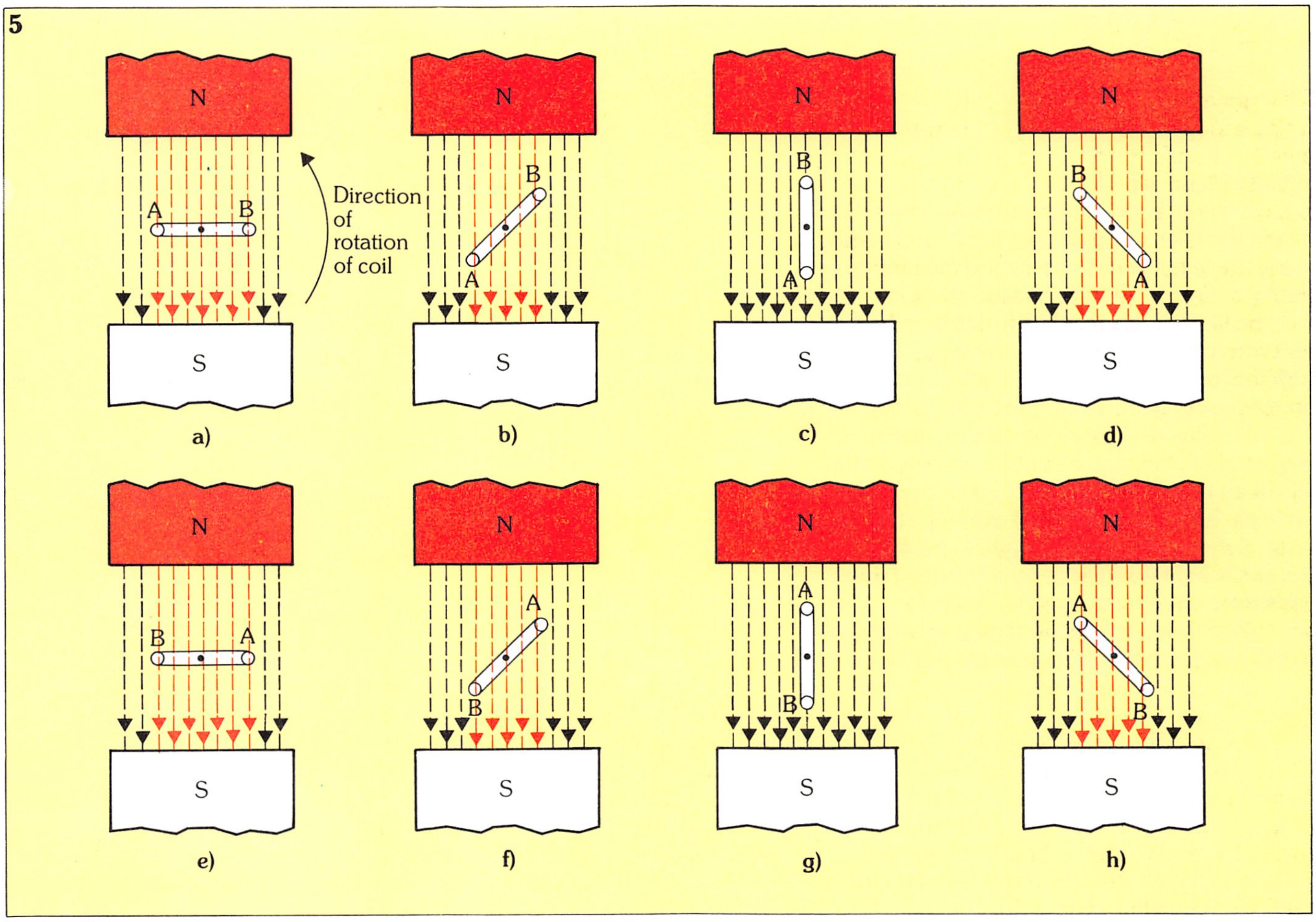
4. A single loop of wire rotating at a constant speed in a uniform magnetic field induces an EMF.

5. One full rotation of the coil showing flux linkages.

4



5



always in such a direction as to oppose the motion or change that produced it.

Calculation of induced voltages

Before leaving this subject, let's work through some examples of the calculations involved. If the conductor, shown in *figure 2*, is 400 mm long and moves through a magnetic field of flux density 0.7 T at a velocity of 25 ms^{-1}, then the EMF generated is:

$$\begin{aligned} E &= Blv \\ &= 0.7 \times 0.4 \times 25 \\ &= 7\,V \end{aligned}$$

Secondly, consider a single loop of wire of area 0.06 m^2, situated in a magnetic field in which the flux density falls from 0.25 T to zero in 10 ms. The flux passing through the loop is:

$$\begin{aligned} \phi &= BA \\ &= 0.25 \times 0.06 \\ &= 0.015\,Wb \end{aligned}$$

The EMF generated is given by:

$$\begin{aligned} E &= -\frac{\phi_2 - \phi_1}{T} \\ &= \frac{0.015}{10 \times 10^{-3}} \\ &= 1.5\,V \end{aligned}$$

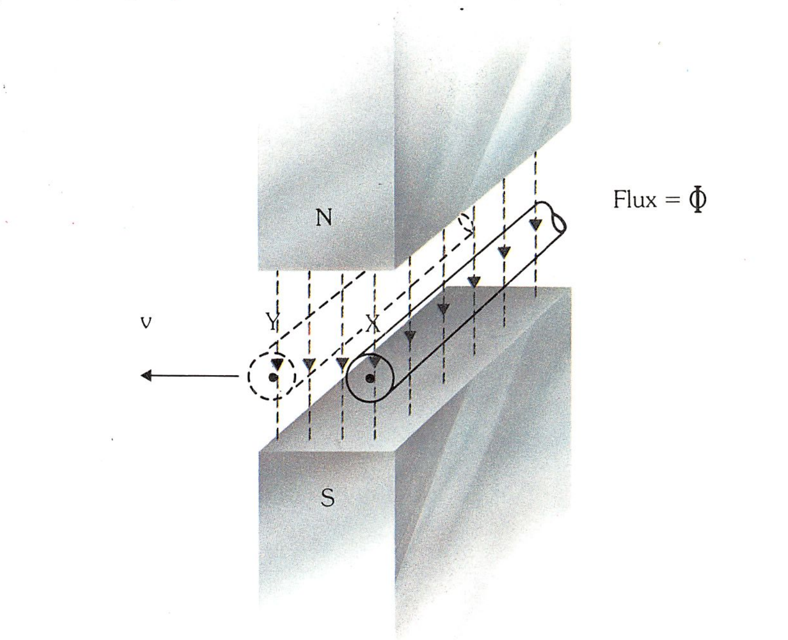
EMF generated in a rotating coil

A single loop of wire rotating at a constant speed in a uniform magnetic field will generate an EMF (*figure 4*). The simplest way to determine the EMF is to draw a graph that shows how the flux linking with the loop varies, as the angle between the flux and the loop changes. *Figure 5* shows one full rotation of the coil; the lines of flux that link with the coil are shown in red. There is a maximum flux linkage with the loop shown in *figure 5a*; it is smaller in *5b*; and in *figure 5c* it has fallen to zero. The flux then changes its sign, and increases in magnitude in the opposite direction until it reaches its maximum negative value in *figure 5e*. This trend continues until the loop returns to its original position (shown in *figure 5a*). The flux linkages at each of these positions of the coil's rotation are shown on the graph in *figure 6a*. This cycle will repeat itself as long as the coil continues to rotate.

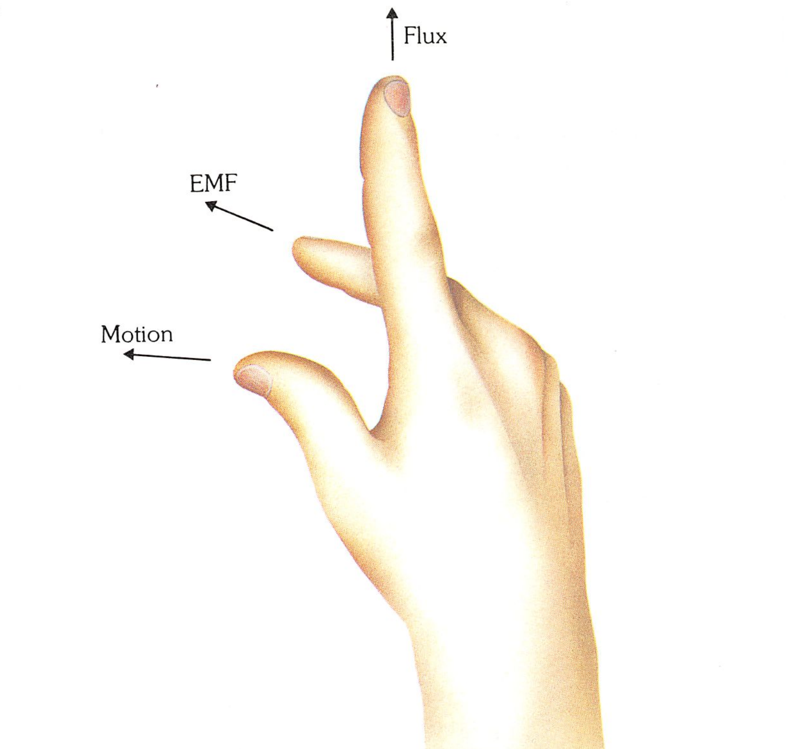
If the wire loop rotates at an angular speed of f revolutions per second, then the time taken for one rotation, T seconds, is equal to 1/f. This is the **period** of the alternating flux and is equal to the time that elapses before any value of flux is repeated. Therefore, when the angle of rotation is 360°, time T has elapsed.

We can use the flux graph to determine the EMF generated. Remember, the EMF is equal to the negative of the rate of change of flux linked. Earlier on, we saw that the rate of change of flux linked can be found by taking very small steps in time and noting the change of flux which occurs during this short interval. If we make the intervals shorter and shorter, we will find that this will ultimately approximate to the shape of the flux curve. The steeper the slope of the curve, the greater the rate of

2



3



2. A single straight wire cutting the lines of flux.

3. Flemings's right-hand rule.

BASIC THEORY REFRESHER

ELECTRICAL TECHNOLOGY

EMF induced in a moving conductor

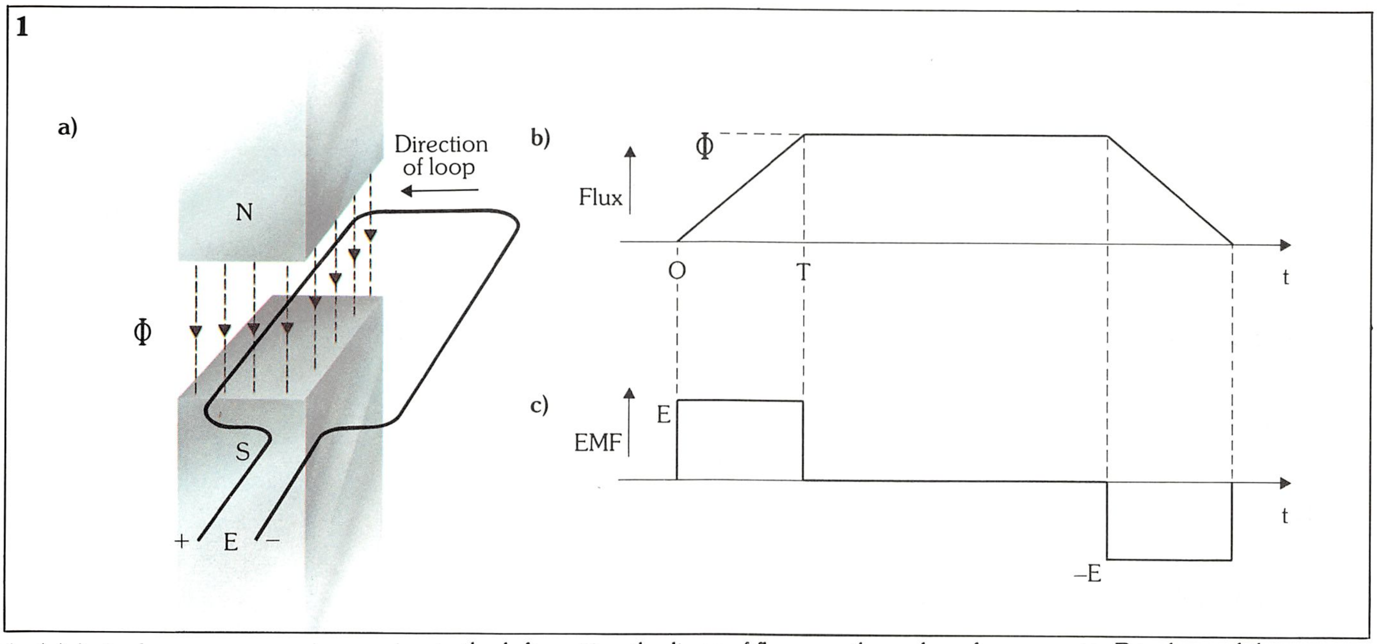
We have seen that any change in the flux linking an electric circuit will generate an EMF. We also know that when this change of 4lux is caused by a changing current in a fixed circuit, then we have the phenomenon of self or mutual induction. Now, what happens when the flux linking a circuit changes, because the circuit itself is moving through a fixed flux?

Coil moving in a magnetic flux

Figure 1a shows a single rectangular loop of wire moving to the left, cutting the lines of flux created by the magnet. During the period of time that we are considering, the right-hand side of the coil does not enter the magnetic field. As the left-hand side passes through the field, it cuts a steadily rising number of lines of flux – increasing to a maximum value ϕ. If the coil moves with a uniform velocity, the flux will increase linearly with time. This is shown in *figure 1b*, where it is assumed that the coil's side takes time, T, to pass between the poles. The changing flux linking with the loop will generate an EMF,E, the magnitude of which is given by:

$$E = -\frac{\phi}{T}$$

If the coil continues to be moved to the left at the same velocity, the right-hand side will eventually cut the lines of flux. When this happens, an EMF of the same magnitude, but of opposite polarity, to the previous EMF will be generated (see *figure 1c*). The flux will, of course, fall to zero when the coil has passed right through the field.



1. (a) A single rectangular loop of wire moving to the left and cutting the magnet's lines of flux; **(b)** graph of flux *vs* time; **(c)** graph of EMF *vs* time.

Flux cutting rule

Instead of a loop, now consider a single straight wire moving across a magnetic flux as show in *figure 2*. This will give us what is known as the **flux cutting rule**: if the flux density between the poles of a magnet is B tesla, and the conductor moves from position X to position Y with a velocity of v metres per second, then the EMF, E volts, generated in a conductor of length l metres is given by:

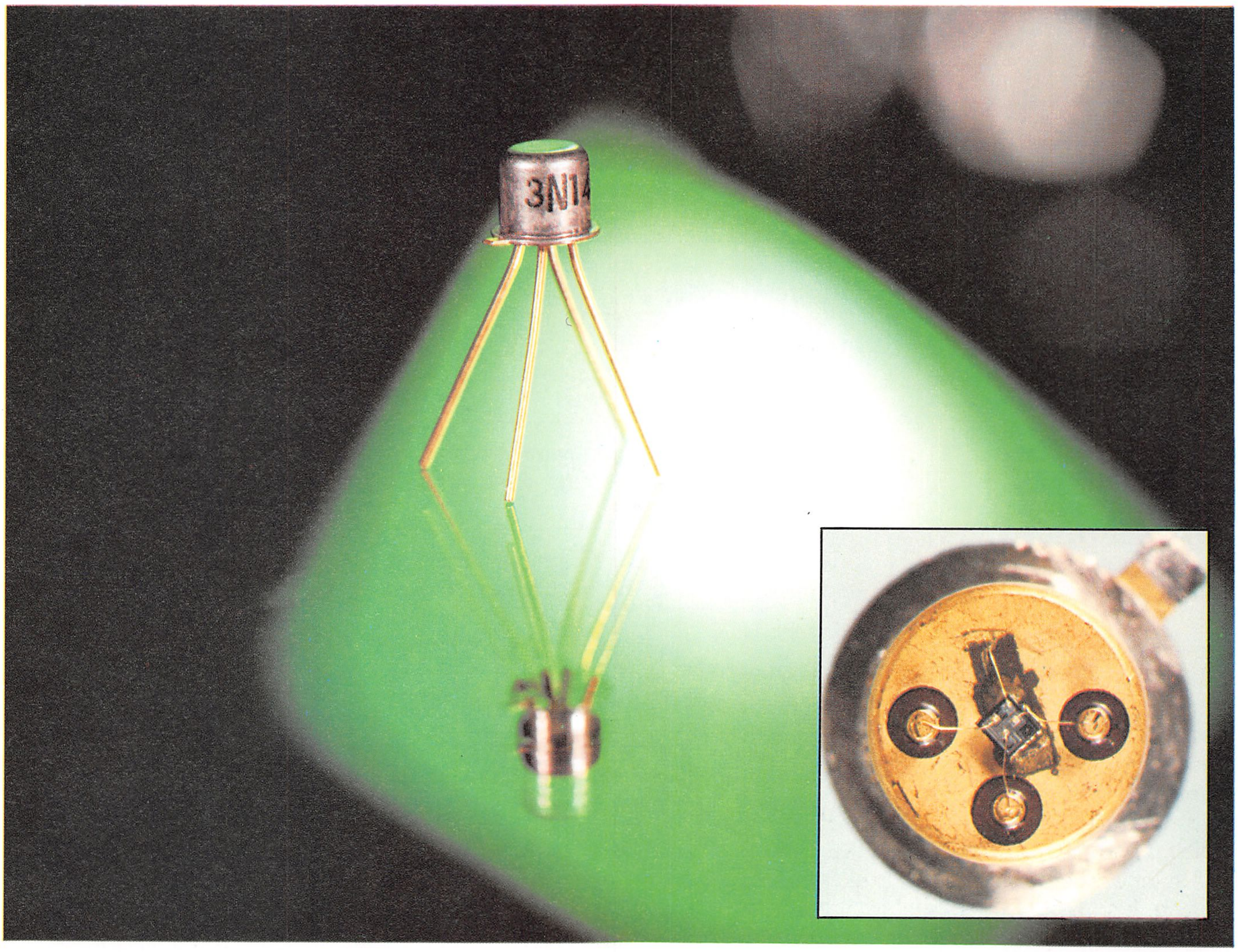
$$E = Blv$$

The direction of the induced EMF can be found by Fleming's **right-hand rule** – which is confusingly similar to his left-hand rule. Hold your right hand, with the thumb, first finger and middle finger at right angles to each other, as shown in *figure 3*. The thu**M**b indicates the conductor's direction of **M**otion, the **F**irst finger the direction of **F**lux, and th**E M**iddle **F**inger points in the direction of the induced **EMF**.

Alternatively, the direction of the generated EMF can be found by Lenz's law, which states that the induced potential difference is

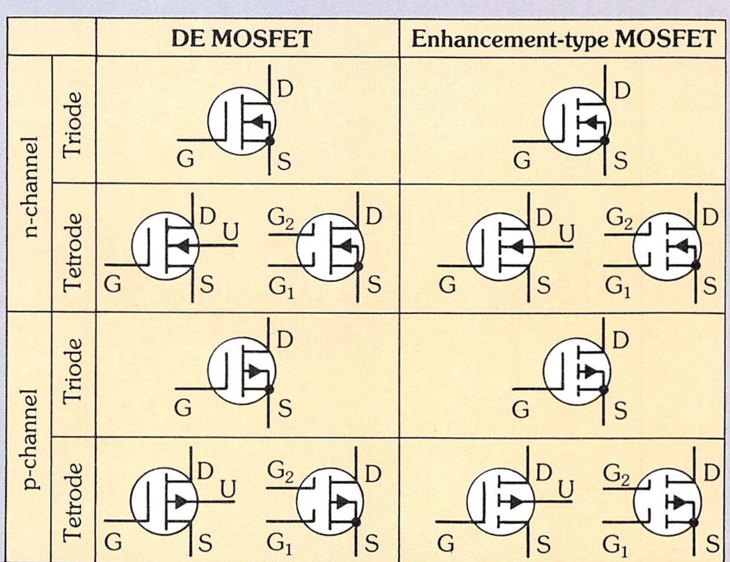
Glossary

DE MOSFET	a depletion/enhancement-type MOSFET formed by heavily doping the MOSFET source and drain areas and lightly doping a channel between the two
deplete	reduction of the number of charge carriers in a semiconductor region so that current flow through the region is decreased
drain, gate, source	three terminals of a FET. A fourth terminal, formed by a second gate, or sometimes a connection to the transistor's substrate, can be made in a MOSFET
dynamic drain resistance, r_d	the reciprocal of the ratio between small changes in drain current and small changes in drain-source voltage, when the gate-source voltage of a FET is held constant
enhance	to increase the number of charge carriers in a semiconductor region so that a corresponding increase in current flow occurs
enhancement-type	a MOSFET in which the channel between source and drain is formed initially of opposite type to the source and drain. Under normal operating conditions the channel-type inverts to be the same as source and drain
inversion layer	MOSFET channel in which the charge carriers are of opposite polarity to the majority carriers of the substrate
MOSFET	insulated gate field effect transistor formed from a layer of semiconductor which is insulated, onto which the gate contact is metallized
n-channel	layer of p-type semiconductor between source and drain of a MOSFET which inverts to n-type under normal operating conditions
p-channel	layer of inverted n-type semiconductor (see n-channel)
saturated	operation of a MOSFET when an increase in drain-source voltage creates minimal increase in drain current
tetrode	a four terminal MOSFET with connections to drain and source, and connections to either two gates or one gate and the substrate
transconductance, g_m	ratio between small change in drain current and the corresponding change in gate-source voltage
triode	a three terminal MOSFET with connections to source, drain and gate



Neville Miles

8



Symbols

The usual symbols for MOSFETs are shown in *figure 8*. DE MOSFETs conduct even with zero gate-source voltage and this is represented by a continuous line between drain and source. Enhancement-type MOSFETs, on the other hand, have no drain current flow for zero gate-source voltage and this is represented by a broken line between drain and source. The horizontal line on the gate is always shown at the transistor's source end.

The substrate is shown as an arrowhead perpendicular to the channel: the direction of the arrow indicating the channel type (i.e. n or p). In some tetrode MOSFETs, the substrate is connected into a circuit and is biased to act like a second gate, In such MOSFETs the substrate has its own terminal connection which is given the symbol, U.

gate-source voltage. It can be represented mathematically by the equation:

$$g_m = \frac{\Delta I_D}{\Delta V_{GS}}$$

when the drain-source voltage is constant. Typical values of MOSFET transconductance are between 500 and 10,000 μS.

A second parameter, the **dynamic drain resistance**, can be defined by holding the gate-source voltage constant such that its reciprocal is the ratio between small variations in drain current and small variations in drain-source voltage, i.e:

$$\frac{1}{r_d} = \frac{\Delta I_D}{\Delta V_{DS}}$$

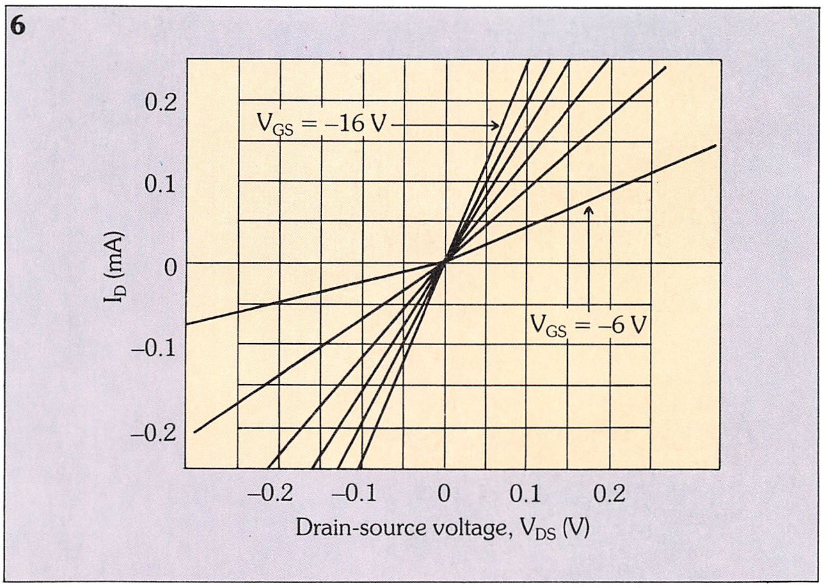
Typical values for dynamic drain resistance are about 20 Ω to 50 kΩ.

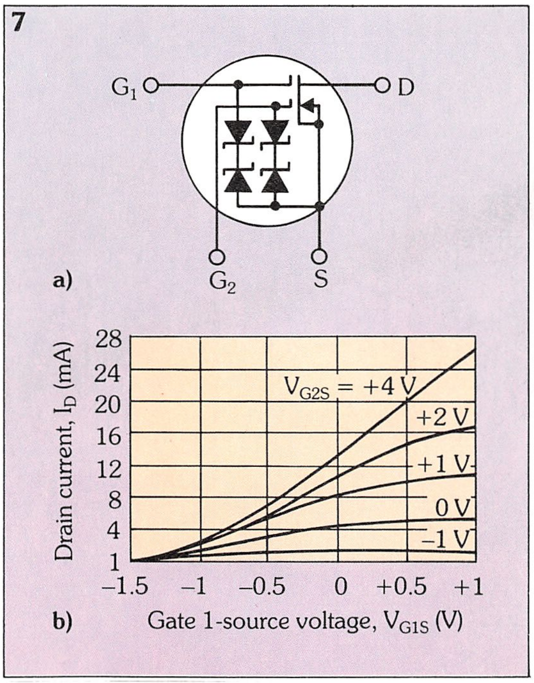
Operation in the non-saturated region

Like JFETs, MOSFETs behave like voltage controlled resistors when the drain-source voltage is lower than the threshold voltage. On the characteristic curves of *figure 4*, this is indicated by the regions below each curve's knee (i.e. the non-horizontal parts). In this non-saturated region, the drain current is proportional to the drain-source voltage, and the resistance between drain and source is controlled by the gate-source voltage. *Figure 6* shows a graph of drain-source voltage against drain current in the non-saturation region, for a number of different values of gate-source voltage.

Double gate MOSFETs

In some applications, notably high frequency amplifiers such as those used as modulators and demodulators in radio and television circuits, it is useful to have two separate input terminals to an amplifier. *Figure 7a* illustrates an n-channel DE MOSFET, the Motorola 3N201, which has two gate terminals, G_1 and G_2. A graph of drain current against the first gate-source voltage variations V_{G1S}, is shown in *figure 7b*. *Figure 7a* shows that Zener diodes are used within the MOSFET to prevent high positive or negative voltages damaging either gate. As the transistor has four terminals it is called a **tetrode** ('tet' meaning four), in contrast with the MOSFETs already covered which are known as **triodes**.





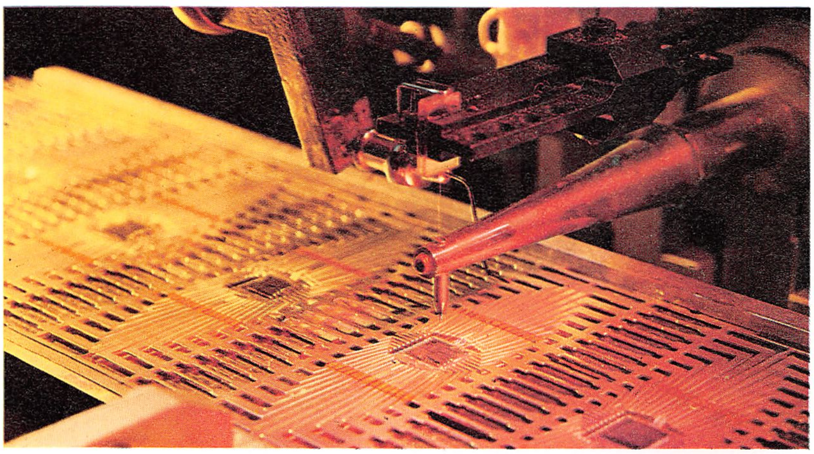
6. I_D *vs* V_{DS} in the non-saturation region for a number of different values of V_{GS}.

7. (a) An n-channel DE MOSFET showing use of Zener diodes; **(b)** graph of I_D *vs* V_{G1S}.

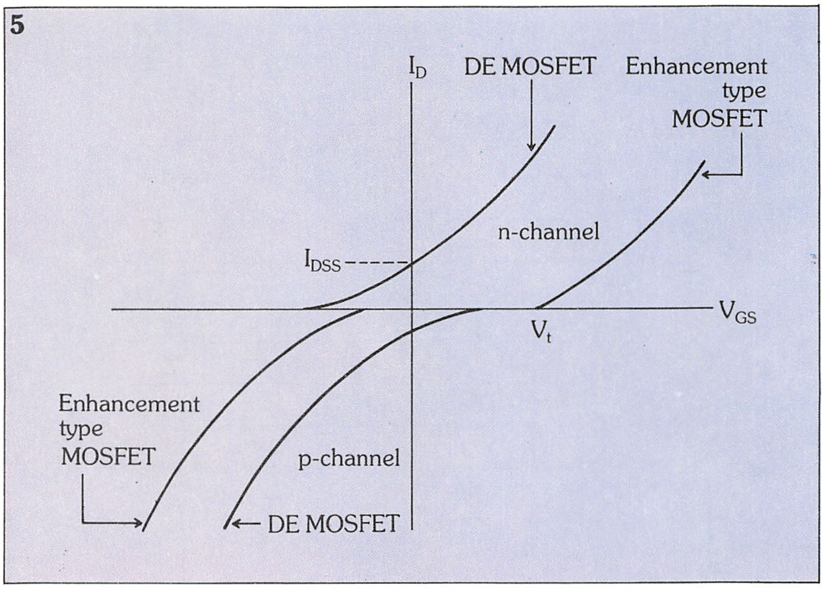
Right: the 3N 141 MOSFET showing: the whole device; the top cut away.

8. Circuit symbols for MOSFETs.

Below: MOSFETs on a silicon chip being wire bonded to package terminals.



5



voltage exceeds the zener breakdown voltage, the diode conducts and prevents further voltage build-up.

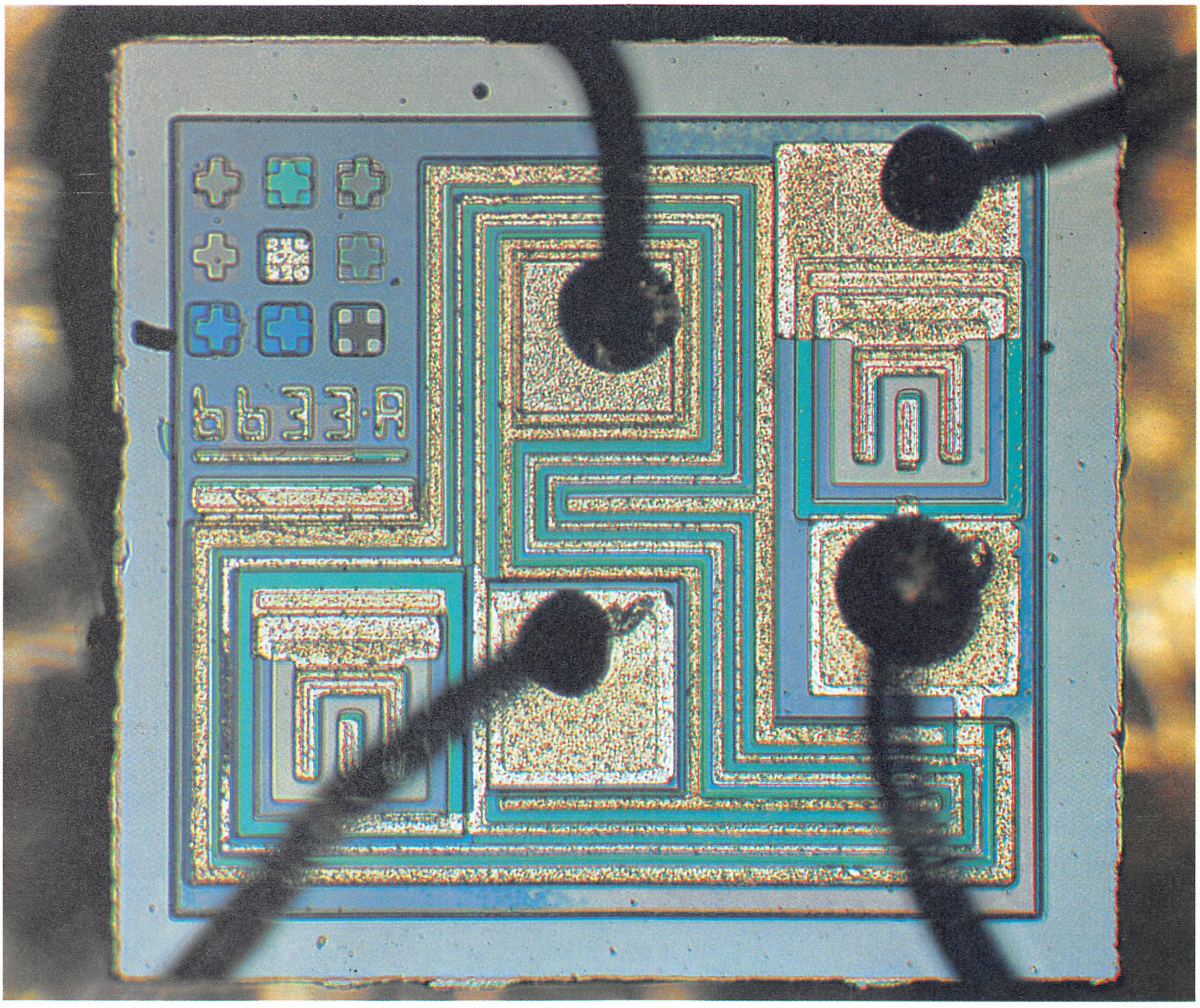
Transconductance

The parameter which indicates MOSFET gain is **transconductance**, g_m. Gain is always defined as output divided by input, and because the output of a MOSFET is its drain current and its input is the gate-source voltage, the unit of transconductance is:

$$\frac{\text{amps}}{\text{volts}} = \frac{1}{\text{ohms}}$$

and is the siemen, S.

Transconductance is defined as the ratio between a small change in drain current and the corresponding change in



Neville Miles

when the gate-source voltage is positive, *the drain current decreases* (due to the channel conductivity being **depleted**), thus it is known as a **depletion/enhancement-type MOSFET** (DE MOSFET or depletion MOSFET).

Characteristic curves

Characteristic curves for both enhancement-type and DE MOSFETs are shown in *figure 4*. Although both sets of curves are similar, the DE MOSFET curves illustrate how either a positive *or* a negative gate-source voltage can be applied to control the drain current. The enhancement-type MOSFET curves show only single polarity negative gate-source voltages.

The transfer curves of *figure 5* show the two types of MOSFET, as well as both n-channel and p-channel varieties; I_{DSS} gives the value of drain current with the gate connected to the source. Each transfer curve crosses the gate-source voltage axis, either to the left or the right of the drain current axis, according to the type of device.

Gate protection in a MOSFET

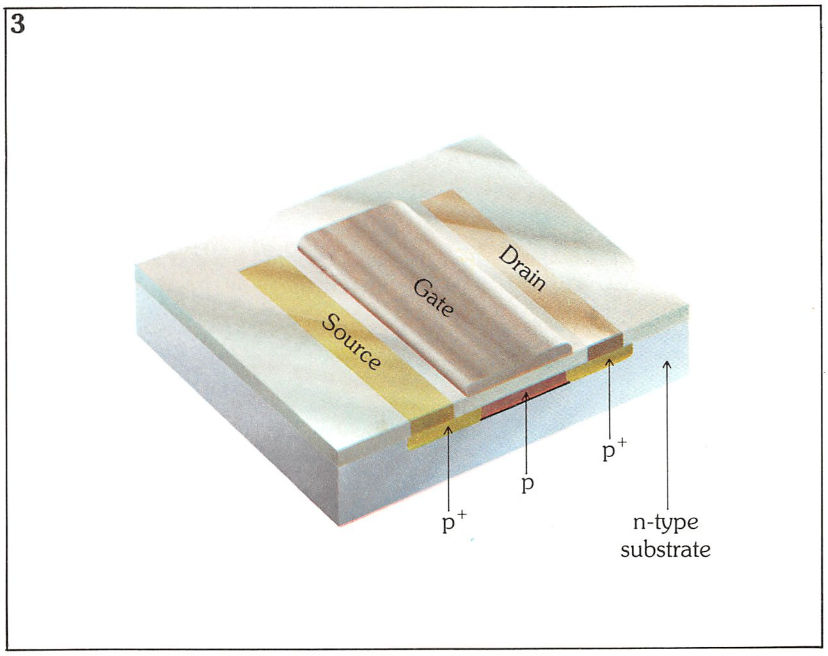
As the MOSFET gate is insulated from the source and drain by a thin layer of silicon dioxide, the transistor can be thought of as a capacitor (i.e. two metal plates separated by a dielectric). The input resistance of a MOSFET is extremely high (between 10^{12} and $10^{15}\ \Omega$), so any charge which may build up on this 'capacitor' produces a voltage, according to the relationship:

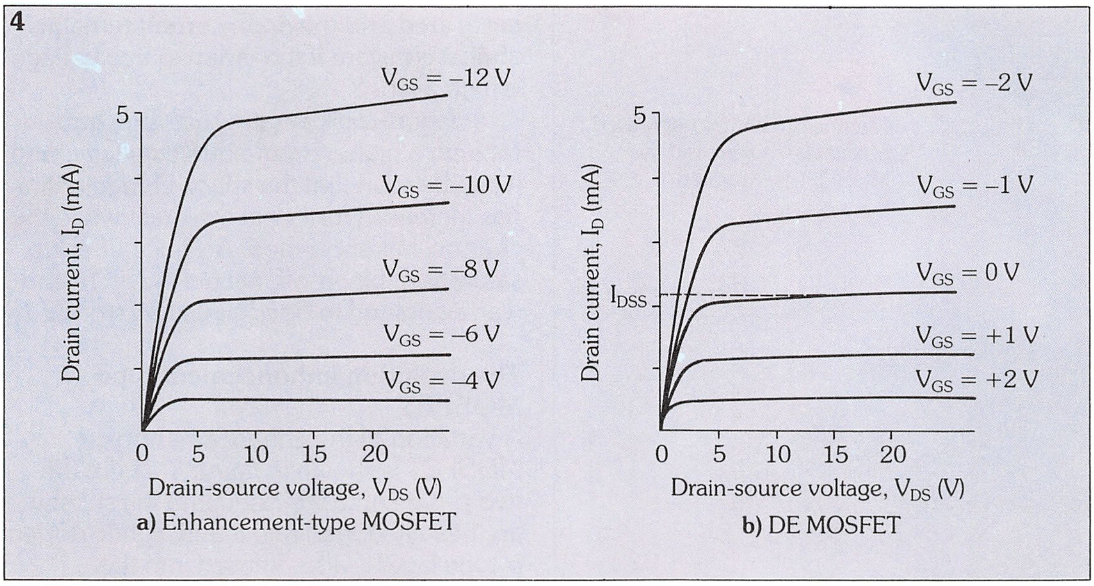
$$V = \frac{Q}{C}$$

If this voltage is higher than the gate can withstand, then the transistor will be destroyed – even the build up of static charge in the human body is sufficient to cause this to happen.

To prevent transistor breakdown manufacturers insert a zener diode between the gate and source. Under normal conditions the zener is off and has no effect, however, as soon as the gate-source

3. Structure of a depletion/enhancement-type MOSFET or DE MOSFET.



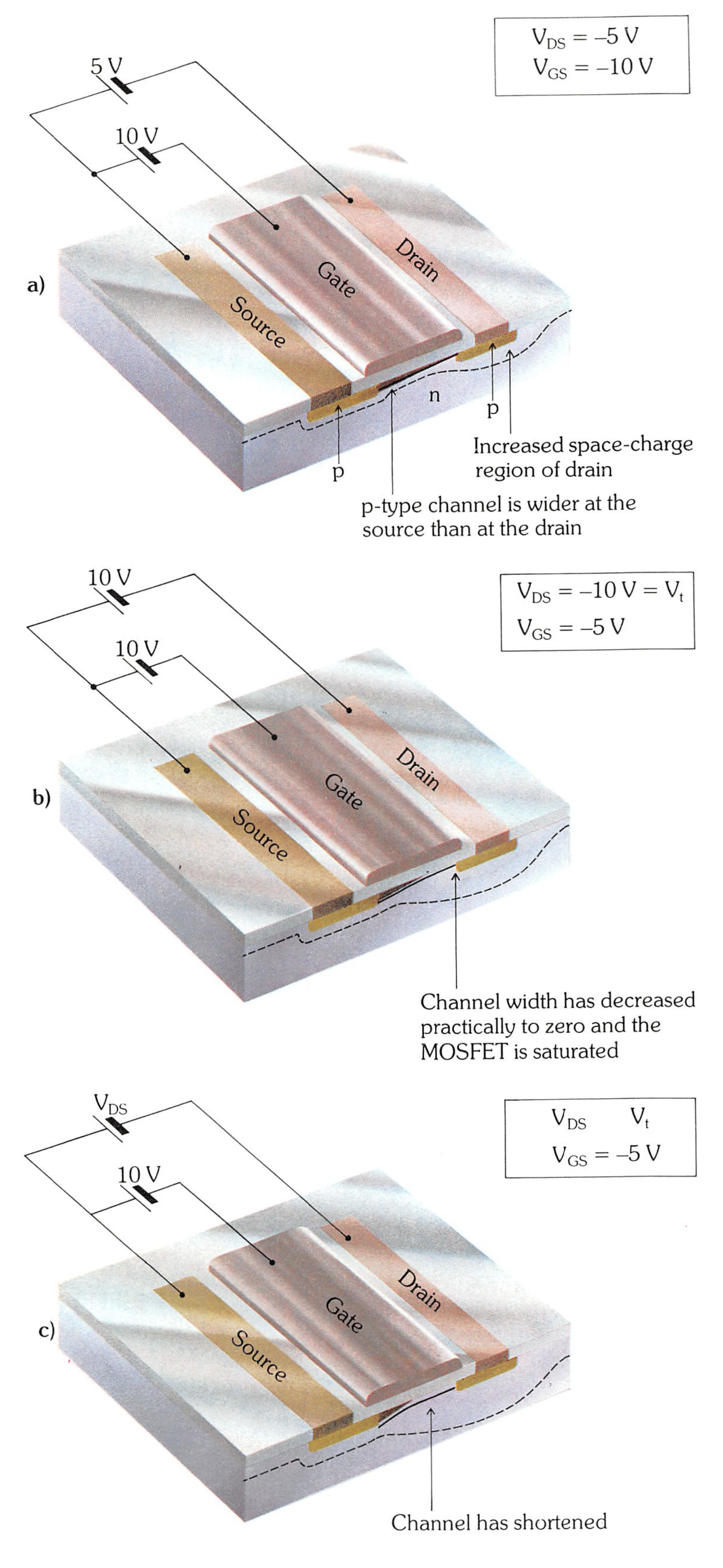


4. Characteristic curves for: (a) enhancement-type MOSFET; and **(b)** DE MOSFET.

5. Drain current *vs* gate-source voltage transfer curves for both n and p-channel DE MOSFETs and enhancement-type MOSFETs.

Right: photomicrograph of a MOSFET.

2



increased, holes are attracted towards the gate to form an **inversion layer** of mobile positive charge carriers, thereby **inverting** the n-type substrate in the channel to p-type between source and drain and creating a p-channel MOSFET. We say that the conductivity of the channel has been **enhanced**, and if a voltage is applied between drain and source, current will flow. The size of this current is controlled by the voltage at the gate, because a change in gate-source voltage will change the number of charge carriers present in the channel.

Figure 2a illustrates the situation with an applied drain-source voltage and a negative gate-source voltage: the voltage between gate and channel now varies along the channel. At the source, the voltage is at maximum (-10 V in *figure 2a*) but falls to a minimium (-5 V) at the drain. The effect of this is to vary the channel width from its maximum at the source, to its minimium at the drain; at the same time, the space-charge region is wider at the drain than at the source.

A situation is shown in *figure 2b* where the drain-source voltage is equal to the transistor's threshold voltage, V_t, and the channel width has decreased almost to zero at the drain. The transistor is said to be **saturated** and the drain current remains almost constant if the drain-source voltage is increased.

Figure 2c illustrates the same transistor with a higher drain-source voltage, and it can be seen that the space-charge region has increased further at the drain while the channel has shortened. A similar effect to saturation, pinch-off, occurs in JFETs and was examined in *Solid State Electronics 13*.

The depletion/enhancement-type MOSFET

A variation of the enhancement-type MOSFET is shown in *figure 3* – here the two p-type areas diffused into the substrate are heavily doped and a thin, lightly doped p-type layer is also diffused into the substrate between the source and drain. This lightly doped p-type channel allows drain current to flow even when the gate-source voltage is at 0 V. When the gate-source voltage is negative, the drain current increases (due to enhancement);

16

SOLID STATE ELECTRONICS

MOSFETs

Introducing MOSFETs

Although early research into the possibility of semiconductor amplifiers was directed towards the field-effect transistor, the discovery that minority carriers could be injected across a p-n junction concentrated attention onto bipolar transistors. Their exceptional characteristics and versatility allowed them to be used in an extremely wide range of applications: both as discrete electronic components and as part of integrated circuits. This, coupled with the fact that no totally adequate manufacturing process was yet available for FETs, meant that bipolar semiconductors rapidly became the more popular.

However, the recent rapid development of manufacturing techniques for surface oxidation of silicon has allowed the mass production of high quality field effect transistors known as **MOSFETs** (metal-oxide semiconductor field effect transistors) or MOSTs (metal-oxide semiconductor transistors). These metal-oxide techniques enable the manufacture of transistors which are many times smaller than bipolar transistors – they are consequently more often used within ICs, allowing considerable complexity within a single IC.

MOSFET structure

The structure of a p-channel MOSFET is shown in *figure 1*. A lightly doped n-type wafer (typical thickness about 150 μm) of silicon is used as the transistor substrate. Two p-type areas are diffused into the substrate 5 μm to 50 μm apart, forming the transistor's source and drain.

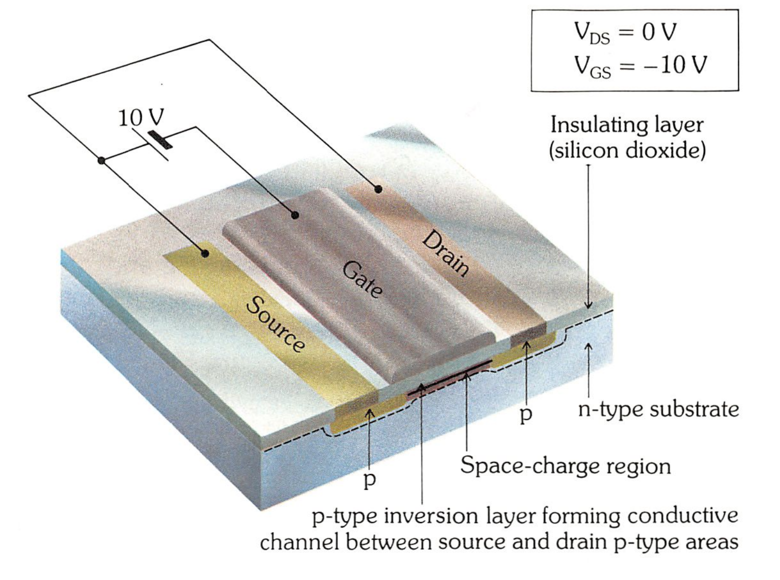
A thin layer of insulating material, generally silicon dioxide, covers the surface of the substrate between the source and drain. Onto this, the transistor's gate contact is metallized using an aluminium vacuum vaporising process; similarly, the source and drain contacts are metallized onto the two p-type diffused areas. This construction produces an insulated gate FET and it is easy to see where its name – MOSFET – originates, i.e. metal (gate) oxide (silicon dioxide) semiconductor.

How a MOSFET works

The transistor shown in *figure 1* is known as a **p-channel enhancement-type MOSFET**: this may seem odd when its channel, the area of substrate between source and drain, is of *n-type silicon*. In order to explain this, we'll look at the operating conditions of the MOSFET.

1. Structure of a p-channel enhancement-type MOSFET.

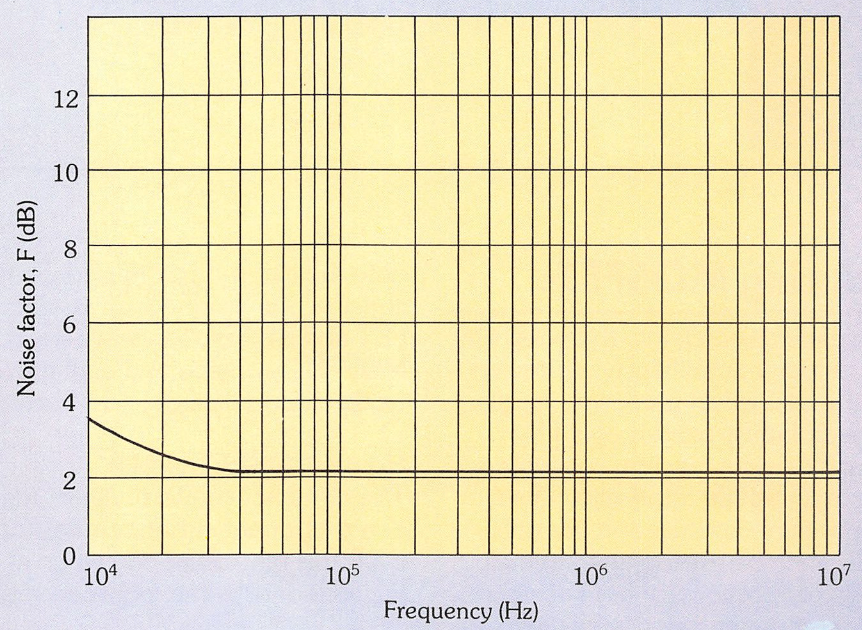
1



Considering the case in *figure 1* where both source and drain are connected to 0 V and the gate contact is connected to −10V. A negative charge accumulates on the gate contact which repels the free electrons in the n-type silicon substrate. A space-charge region therefore exists between the gate and substrate and this spreads under the source and drain. If the negative gate voltage is

2. The effect on the channel width of a p-channel enhancement-type MOSFET when:
(a) a drain-source voltage is applied;
(b) the drain-source voltage is equal to the transistor's threshold voltage;
(c) a higher drain-source voltage is applied.

2



2. The noise factor of a typical FET.

giving the ideal case where F = 0 dB; in practical applications semiconductor noise factors from 3 to 10 dB are common.

Noise in transistors

A transistor's noise factor is defined at specified conditions of resistance, frequency, bias voltage and current, and depends upon the physical characteristics of the transistor.

The graph in *figure 1* illustrates the noise factor of a typical bipolar transistor, with a collector current of 2 mA and collector-emitter voltage of 6 V, as a function of the operating frequency.

Similarly, the graph in *figure 2* illustrates how the noise factor of a typical FET varies with operating frequency. The generally lower noise factor displayed by the FET is one reason why FETs are often used to amplify very small signals, where the addition of only a tiny amount of noise would have a significant effect on the signal-to-noise ratio. After the signal has been amplified by a FET, bipolar transistors can be used as amplifiers, because the noise they add will be smaller in comparison with the already amplified signal.

Glossary

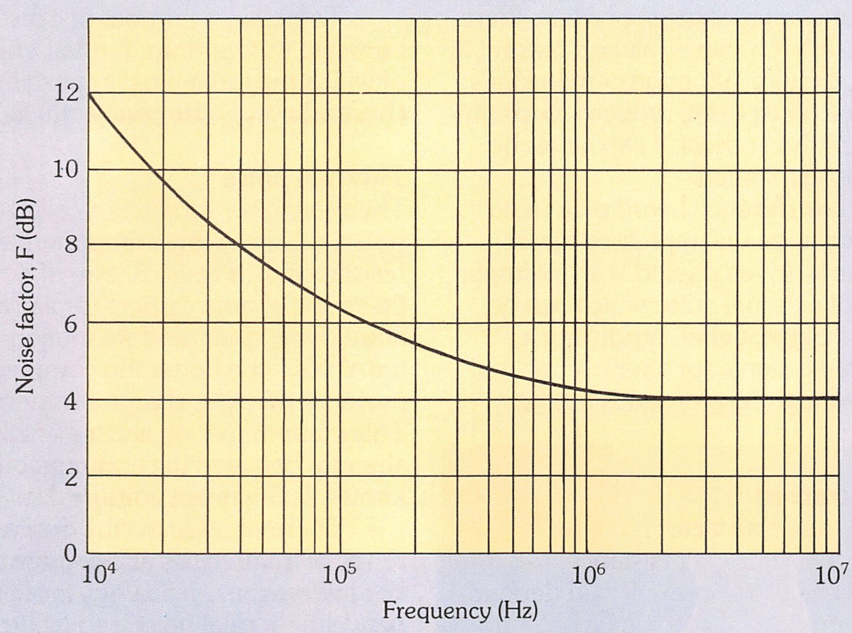
fundamental noise	noise generated within the electronic components of a circuit. There are three main types: flicker, shot and thermal noise
noise	any unwanted signal in an electronic circuit. The noise can be produced by external (interference) or internal factors (fundamental noise)
noise factor, F	the ratio of the input signal-to-noise to the output signal-to-noise. An ideal circuit has a noise factor of 1 (i.e. 0 dB)
signal-to-noise ratio	the ratio of a circuit's signal power to its noise power
white noise	noise which has a constant power level at all frequencies

carriers within the semiconductor. The **noise current**, I_n, caused by this random flow of charge carriers can be calculated from the following expression:

$$I_n^2 = 2qIB$$

where: q is the carrier charge (which, for an electron, is 1.6019×10^{-19} C); I is the size of the DC current upon which the shot noise is superimposed; and B is the system bandwidth.

1



1. The noise factor of a typical bipolar transistor as a function of the operating frequency.

Flicker noise

Flicker noise (also called **excess noise** or **1/f noise**) unlike thermal and shot noise is not white noise. The amount of flicker noise within a circuit is inversely proportional to frequency, therefore it is greatest at lower frequencies.

The noise current of flicker noise at a particular frequency is very approximately given by the expression:

$$I_n^2 = \frac{B}{f}$$

where: B is the system bandwidth; and f is the frequency.

Noise factor

The amount of noise present in a complete electronic signal is often specified as a ratio of the signal power and the noise power. This **signal-to-noise ratio** is generally quoted in decibels (dB) and calculated from the expression:

$$10 \log\left(\frac{\text{signal power}}{\text{noise power}}\right)$$

If a signal is applied to the input of, say, a transistor amplifier, it will be amplified and the output signal will therefore be a larger version of the input signal. With an ideal amplifier no noise would be added, so the output signal-to-noise ratio would be the same as the input signal-to-noise ratio. However, as electronic components all produce fundamental noise – thermal, shot and flicker – the output signal will have a lower signal-to-noise ratio than the input signal-to-noise ratio.

The **noise factor**, or **noise figure** F, of a device is simply the ratio of input signal-to-noise to output signal-to-noise, i.e:

$$F = \frac{\text{input signal-to-noise}}{\text{output signal-to-noise}}$$

For the ideal amplifier the two signal-to-noise ratios are the same, so the noise factor is 1, however, in all practical situations the noise factor will be greater than 1. Generally, though, the noise factor is expressed in decibels, using the expression:

$$10 \log F$$

15
SOLID STATE ELECTRONICS

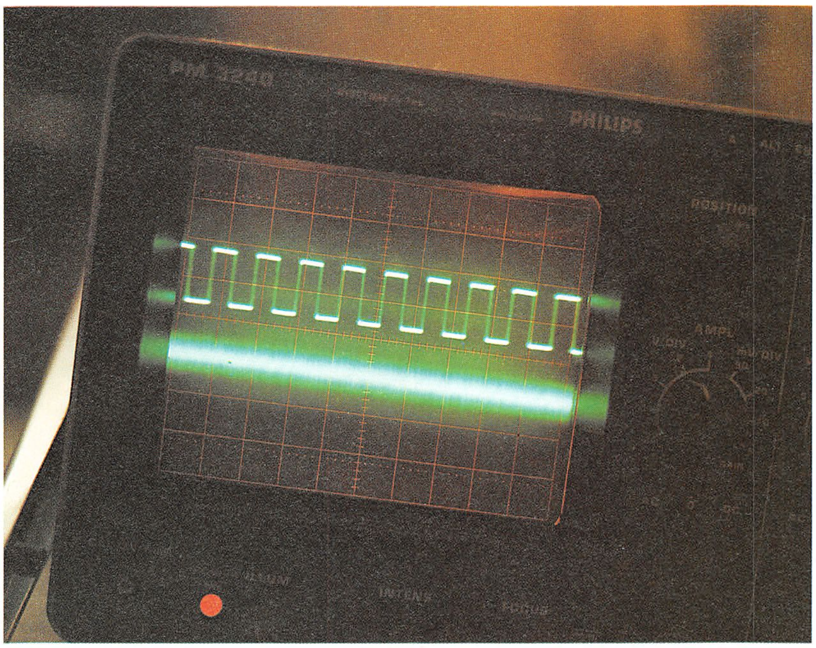
Noise in semiconductor components

What is noise

In electronics, the term **noise** is used to describe unwanted signals, of which there are many different types, always present in electronic circuits. Although certain kinds of noise are more easily reduced to insignificant levels than others, it can never be completely eliminated.

The **interference** heard on a radio receiver when an unsuppressed car or motor bike is driven passed is an example of a type of external noise which *can* be reduced: a bigger aerial, producing a stronger radio signal (or a new set of suppressed high-tension leads for the offending engine) will prevent the interference. However, even when all possible sources of interference which can be eliminated are removed, there will still be noise. For example, the background *hiss* which can often be heard between sections of music on a radio receiver is the effect of noise being generated *within* the electronic components of the radio circuit. This internal noise is often referred to as **fundamental noise**.

Below: an oscilloscope illustrating random noise in a digital circuit. The fuzzy bottom signal would appear as a straight line if there was no noise.



Paul Brierley

There are a number of causes contributing to this random, fundamental noise of which the three most important are: **thermal noise**; **shot noise**; **flicker noise**.

Thermal noise

Thermal noise, sometimes called **Johnson noise**, occurs in any component which has resistance. In a resistor, even if no current flows, the charge carriers display a temperature dependent random motion between terminals – the hotter the component, the more random the charge carrier motion. This random motion causes small voltage changes between the component terminals known as the **noise voltage**, V_n.

Thermal noise occurs over a wide range of frequencies at a constant level. For this reason, an analogy is often made between thermal noise and white light (white light being a combination of equal levels of all colours of light) and consequently thermal noise is often termed **white noise**.

We can calculate the value of the noise voltage, V_n, using the following expression:

$$V_n^2 = 4kTRB$$

where: k is Boltzmann's constant ($k = 1.38 \times 10^{-23}\ JK^{-1}$); T is the absolute temperature (expressed in K); R is the component's resistance (expressed in ohms); and B is the bandwidth of the system (expressed in Hz).

Thermal noise often occurs in resistive elements of a circuit, including those parts of semiconductor devices which act as resistors.

Shot noise

Shot noise is only found in semiconductors and, like thermal noise, is also white noise. It is a result of the irregular flow of charge

pulses at a precise 20 kHz frequency. These are counted by a modulo-20,000 counter, which therefore gives a single output pulse every second. A series of three other counters count seconds, minutes and hours, each using the appropriate modulus.

The parallel binary output of each counter is decoded to drive a seven-segment display for each digit of hours, minutes and seconds. This circuit does not need synchronous counters, because the rippling of states takes place so fast that the eye would not see it on the display, and also, the division accuracy is not affected by rippling in this circuit.

Other types of structural units

There are many other different types of counters. For example, some counters can be preset to a particular state, in the same way that parallel data can be entered in a register. Others include decoders in their parallel outputs, which generate an output at a particular count. What we have seen so far though, is enough to give you a general idea of how counters work.

You should also have a good general picture of all the different building blocks by now. The sequential and combinational units looked at in this chapter and the last one, represent all the types of digital structural units which you will come across. Other more complicated types can always be understood in terms of these simple units.

To sum up, you can see that we have come a long way from the simple logic gates of the early chapters. We have discovered how to analyse circuits, and how circuits can be built from combinations of gates to perform specific functions. We now know how to connect building blocks together to make memory units, and how to use these memory units to store sequential information.

The different applications of digital systems we have looked at will be further investigated later on, when we will see how the by now familiar structural units are put together to build more complex systems.

Glossary

clear	asynchronous input that returns a flip-flop's output to logic state 0. Also known as reset
clock	regular pulse generator used to control and synchronise digital circuitry. Some flip-flops use it as a data control signal
counter	register made up of flip-flop circuits which has one input and (usually) a parallel output from each flip-flop. Counts pulses arriving at the input and stores the total count in a (usually binary) code
D flip-flop	a clocked flip-flop with one data input (D) whose true output changes to the D state, during the clock signal
J-K flip-flop	a flip-flop with inputs J and K. Acts as an R-S flip-flop, except that when J and K are 1, it will toggle to the opposite state, rather than an unknown state like an R-S flip-flop
modulus	the modulus of a counter is the number of states it counts through before returning to the beginning state. Written as 'modulo' when used as a prefix, as in modulo-12 counter
R-S flip-flop	flip-flop in which a momentary 1 at the R (reset) input, changes the true output to 0, and a momentary 1 at the S (set) input changes the true output to 1
T flip-flop	flip-flop that toggles the outputs to the opposite state when a 1 is at the T (toggle) input

(continued from part 15)

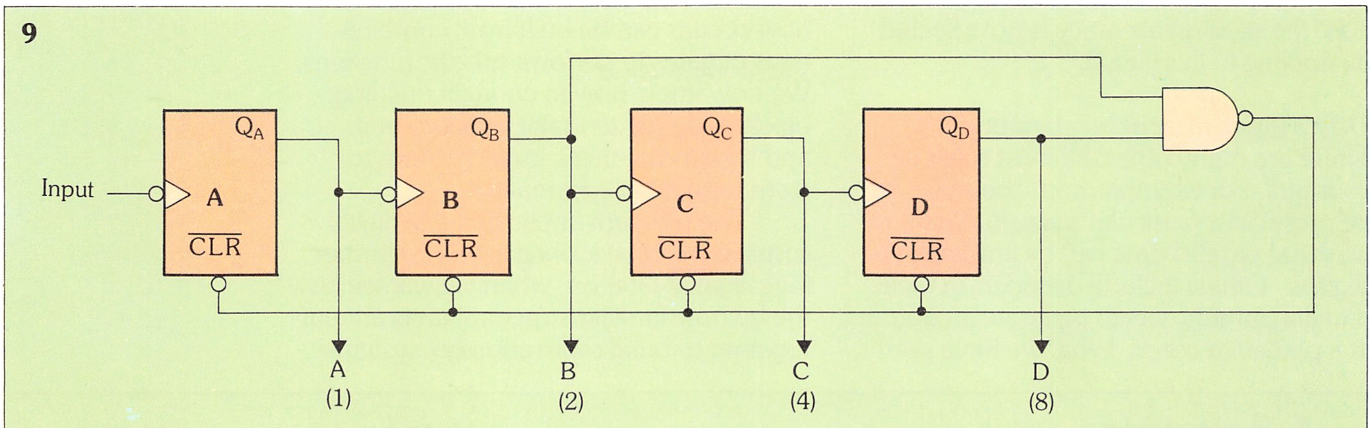
Figure 9 shows how to make a counter which can count from 0 to 9 and back to 0, using binary code. This is a decade (decimal) counter, otherwise called a BCD counter, or a modulo-10 counter.

As you can see, this counter is similar to the 4-bit binary counter in *figure 7*. The steering circuit however, is different; it makes the counter return to 0 after having counted up to nine. This means that the counter **modulus** has been reduced from 16 to 10. The modulus of a counter is equivalent to the number of different states which the outputs can assume during the count. With the proper steering circuitry and enough flip-flops you can make a counter for any modulus that a particular application may require.

Counters in a digital clock

Figure 10 shows how counters can be used in a digital clock to show hours, minutes and seconds in a 12 hour cycle. A quartz oscillator circuit produces sharp, square

9. A modulo-10 counter.



Asynchronous decade counter using forced reset (clear) to zero when count = 0

10. How counters are used in a digital clock to show hours, minutes and seconds.